

Lecture Notes in Computer Science

1503

Giorgio Levi (Ed.)

Static Analysis

5th International Symposium, SAS'98
Pisa, Italy, September 1998
Proceedings



Springer

Springer

Berlin

Heidelberg

New York

Barcelona

Budapest

Hong Kong

London

Milan

Paris

Singapore

Tokyo

Giorgio Levi (Ed.)

Static Analysis

5th International Symposium, SAS'98
Pisa, Italy, September 14-16, 1998
Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Giorgio Levi
Università di Pisa, Dipartimento di Informatica
Corso Italia, 40, I-56125 Pisa, Italy
E-mail: levi@di.unipi.it

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Static analysis : 5th international symposium ; proceedings / SAS
'98, Pisa, Italy, September 14-16, 1998. Giorgio Levi (ed.). - Berlin
; Heidelberg ; New York ; Barcelona ; Budapest ; Hong Kong ;
London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1998
(Lecture notes in computer science ; Vol. 1503)
ISBN 3-540-65014-8

CR Subject Classification (1991): D.1, D.2.8, D.3.2-3, F.3.1-2, F.4.2

ISSN 0302-9743

ISBN 3-540-65014-8 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1998
Printed in Germany

Typesetting: Camera-ready by author
SPIN 10638978 06/3142 - 5 4 3 2 1 0 Printed on acid-free paper

Foreword

This volume contains the proceedings of the 1998 international symposium on static analysis (SAS'98) which was held in Pisa (Italy), on September 14-16, 1998 and was part of a federated conference with ALP-PLILP'98 and several workshops. SAS'98 is the annual conference and forum for researchers in all aspects of static analysis. It follows to SAS'94, SAS'95, SAS'96 and SAS'97 which were held respectively in Namur (Belgium), Glasgow (UK), Aachen (Germany) and Paris (France), and the international workshops WSA'92 held in Bordeaux (France) and WSA'93 held in Padova (Italy).

In response to the call for papers, 48 papers were submitted. All papers were reviewed by at least three reviewers and the program committee met in Pisa to select 20 papers based on the referee reports. There was a consensus at the meeting that the technical papers were of very high quality. In addition to the submitted papers, SAS'98 had a number of outstanding invited speakers. Roberto Giacobazzi, Peter Lee, Amir Pnueli, Dave Schmidt, Scott Smolka, and Bernhard Steffen accepted our invitation to give invited talks or tutorials. Some of the papers (or abstracts) based on these talks are also included in this volume.

SAS'98 has been fortunate to rely on a number of individuals and organizations. I want to thank all the program committee members and referees, for their hard work in producing the reviews and for such a smooth and enjoyable program committee meeting. Special thanks go to the conference chairman, Maurizio Gabbrielli, and to my students in Pisa who helped me a lot. More special thanks go to Vladimiro Sassone, who made available to SAS'98 his excellent system for handling submissions and reviews on the web, and to Ernesto Lastres and Renè Moreno who were my "system managers". The use of this system made my life of program chairman much easier and I strongly recommend it to future program chairpersons.

SAS'98 was sponsored by Università di Pisa, Compulog Network, Consiglio Nazionale delle Ricerche, CNR-Gruppo Nazionale di Informatica Matematica, Comune di Pisa, and Unione Industriali di Pisa.

Giorgio Levi

July 1998

Program Committee of SAS'98

Alex Aiken (Berkeley, USA)
Maurice Bruynooghe (Leuven, Belgium)
Michael Codish (Ben Gurion, Israel)
Agostino Cortesi (Venezia, Italy)
Radhia Cousot (Polytechnique Paris, France)
Alain Deutsch (INRIA, France)
Laurie Hendren (McGill, Canada)
Fritz Henglein (DIKU, Denmark)
Thomas Jensen (IRISA/CNRS, France)
Alan Mycroft (Cambridge, UK)
Flemming Nielson (Aarhus, Denmark)
Thomas Reps (Wisconsin, USA)
Dave Schmidt (Kansas State, USA)
Mary Lou Soffa (Pittsburgh, USA)
Harald Søndergaard (Melbourne, Australia)
Bernhard Steffen (Passau, Germany)

Conference Chairman of SAS-ALP-PLILP'98

Maurizio Gabbrielli (Pisa, Italy)

List of Referees

Torben Amtoft	Andrew Appel	Roberto Bagnara
Dante Baldan	Bruno Blanchet	Rastislav Bodik
Andrew Bromage	Nicoletta Cocco	Thomas Conway
Patrick Cousot	Bart Demoen	Danny De Schreye
Alessandra Di Pierro	Manuel Faehndrich	Gilberto Filé
Jeff Foster	Pascal Fradet	Maurizio Gabbrielli
Etienne Gagnon	Roberto Giacobazzi	Robert Gluck
Eric Goubault	Susanne Graf	James Harland
Nevin Heintze	Frank Huch	Jesper Jorgensen
Andy King	Jens Knoop	Laura Lafave
Vitaly Lagoon	Martin Leucker	Michael Leuschel
Daniel Le Metayer	Henning Makholm	Elena Marchiori
Kim Marriott	Bern Martens	Markus Mueller-Olm
Anne Mulkers	Hanne Riis Nielson	Thomas Noll
Dino Pedreschi	Francesco Ranzato	Jakob Rehof
Olivier Ridoux	Eva Rose	Sabina Rossi
Oliver Ruething	David Sands	Fausto Spoto
Peter Stuckey	Zhendong Su	Cohavit Taboch
Simon Taylor	Peter Thiemann	Arnaud Venet
Zhe Yang	Phillip Yelland	

Contents

Data-Flow Analysis

Bidirectional Data Flow Analysis in Code Motion: Myth and Reality	1
<i>Oliver Rüthing</i>	
On a New Method for Dataflow Analysis of Java Virtual Machine Subroutines	17
<i>Masami Hagiya, Akihiko Tozawa</i>	
Enabling Sparse Constant Propagation of Array Elements via Array SSA Form	33
<i>Kathleen Knobe, Vivek Sarkar</i>	
Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses	57
<i>Michael Hind, Anthony Pioli</i>	

Logic Programming

Analysis of Normal Logic Programs	82
<i>François Fages, Roberta Gori</i>	
The Correctness of Set-Sharing	99
<i>Patricia M. Hill, Roberto Bagnara, Enea Zaffanella</i>	
Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing	115
<i>Valérie Gouranton</i>	

Concurrency

A Symbolic Semantics for Abstract Model Checking	134
<i>Francesca Levi</i>	
Automatic Determination of Communication Topologies in Mobile Systems	152
<i>Arnaud Venet</i>	
Constructing Specific SOS Semantics for Concurrency via Abstract Interpretation	168
<i>Chiara Bodei, Pierpaolo Degano, Corrado Priami</i>	

Abstract Domains

A First-Order Language for Expressing Aliasing and Type Properties of Logic Programs	184
<i>Paolo Volpe</i>	
Refining Static Analyses by Trace-Based Partinoning Using Control Flow ..	200
<i>Maria Handjieva, Stanislav Tzolovski</i>	
Building Complete Abstract Interpretations in a Linear Logic-Based Setting	215
<i>Roberto Giacobazzi, Francesco Ranzato, Francesca Scozzari</i>	

Partial Evaluation

On the Power of Homeomorphic Embedding for Online Termination	230
<i>Michael Leuschel</i>	
Analysis of Imperative Programs through Analysis of Constraint Logic Programs	246
<i>Julio C. Peralta, John P. Gallagher, Hüseyin Sağlam</i>	
Improving Control in Functional Logic Program Specialization	262
<i>E. Albert, M. Alpuente, M. Falaschi, P. Julián, G. Vidal</i>	

Type Inference

Directional Type Inference for Logic Programs	278
<i>Witold Charatonik, Andreas Podelski</i>	
Finite Subtype Inference with Explicit Polymorphism	295
<i>Dominic Duggan</i>	

Optimization

Sparse Jacobian Computation in Automatic Differentiation by Static Program Analysis	311
<i>M. Tadjouddine, F. Eyssette, C. Faure</i>	
A New Solution to the Hidden Copy Problem	327
<i>Deepak Goyal, Robert Paige</i>	

Tutorials

A Tutorial on Domain Theory in Abstract Interpretation	349
<i>Roberto Giacobazzi</i>	

Program Analysis as Model Checking of Abstract Interpretations	351
<i>David Schmidt, Bernhard Steffen</i>	

Invited Talks

Certifying, Optimizing Compilation	381
<i>Peter Lee</i>	

Author Index	383
-------------------------------	-----

Bidirectional Data Flow Analysis in Code Motion: Myth and Reality

Oliver Rüthing

Department of Computer Science, University of Dortmund, Germany
ruething@ls5.cs.uni-dortmund.de

Abstract. Bidirectional data flow analysis has become the standard technique for solving bit-vector based code motion problems in the presence of critical edges. Unfortunately, bidirectional analyses have turned out to be conceptually and computationally harder than their unidirectional counterparts. In this paper we show that code motion in the presence of critical edges can be achieved without bidirectional data flow analyses. This is demonstrated by means of an adaption of our algorithm for lazy code motion [15], which is developed from a fresh, specification oriented view. Besides revealing a better conceptual understanding of the phenomena caused by critical edges, this also settles the foundation for a new and efficient hybrid iteration strategy that intermixes conventional round-robin iteration with the exhaustive iteration on critical subparts.

1 Motivation

In data flow analysis equation systems involving bidirectional dependencies, i. e. dependencies from predecessor nodes as well as from successor nodes, are a well-known source for various kinds of difficulties. First, bidirectional equation systems are conceptually hard to understand. Mainly, this is caused by the lack of a corresponding operational specification like it is given by the the meet over all path (MOP) solution of a uni-directional data flow problem. Furthermore, Khedker and Dhamdhere recently proved that the costs for solving bidirectional data flow analysis problems may be significantly worse than for solving their unidirectional counterparts. This particularly holds for the only practically relevant class of bidirectional analyses, bit-vector based code motion problems. In fact, all known bidirectional problems are of this kind. Even more specifically, they are more or less variations of Morel's and Renvoise's pioneering algorithm for the elimination of partial redundancies [18,12,13,8,11,23,4,9]. Independently different researchers documented that bidirectionality is only required in programs that have *critical edges* [7,15], i. e. edges in a flow graph that directly lead from branch nodes to join nodes (see Fig. 1a for illustration). Ideally, critical edges can be completely eliminated by inserting empty synthetic nodes as depicted in Fig. 1b. In this example, the additional placement point enables the code motion transformation shown in Fig. 1c which eliminates the partial redundant

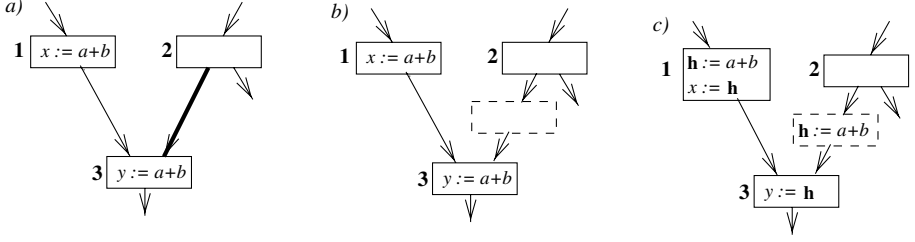


Fig. 1. a) Critical Edge b) Edge splitting c) Transformational gain through edge splitting

computation on the path through node 1 and 3¹ However, in practice splitting of critical edges is sometimes avoided since it may cause additional unconditional jumps or decrease potential for pipelined execution.²

In this paper we investigate a new approach to code motion in the presence of critical edges. This is demonstrated by presenting a “critical” variant of our algorithm for lazy code motion [15]. However, the principal ideas straightforwardly carry over to all related code motion algorithms that employ bidirectional data flow analyses.

Our algorithm is developed from a rigorous, specification oriented view. This particularly allows us to separate between different concerns. While safety in code motion is naturally associated with forward and backward oriented propagation of information, the presence of critical edges requires to impose additional homogeneity properties which can be expressed in terms of a side propagation of information. Actually, this clear separation allows us to avoid the usage of bidirectional dependencies in our specification. With regard to the variant of lazy code motion the contribution of this paper is threefold:

- On a conceptual level we give a unidirectional specification of the problem. This particularly induces the first MOP characterization of code motion in the presence of critical edges.
- We present a novel hybrid iteration strategy that separates the information flow along critical edges from the information flow along the uncritical ones. While the latter one is accomplished by an outer schedule proceeding in standard round-robin discipline the critical information flow is treated exhaustively by an inner schedule.
- Almost as a by-product we obtain the first lifetime optimal algorithm for partial redundancy elimination in the presence of critical edges.

¹ This is not possible in Fig. 1a, since hoisting $a + b$ to node 2 introduces a new value on the rightmost path.

² Sometimes critical edges are not split only in situations that may harm the final code generation.

1.1 Related Work

As Khedker and Dhamdhere [14] and more recently Masticola et al. [17] noticed, critical edges do not add to the worst-case time complexity of iterative data flow analyses being based on a workset approach. However, this result cannot be generalized to bit-vector analyses where the iteration order has to be organized in a way such that structural properties of the flow graph are exploited in order to take maximum benefit of bit-wise parallel updates through efficient bit-vector operations.

Hecht and Ullman [10] proved an upper bound on the number of round-robin iterations that are necessary for stablization of monotone, unidirectional bit-vector problems. When proceeding in reverse postorder (or postorder for backward problems) $d + 2$ round-robin iterations are sufficient where d is the *depth* of the flow graph, i. e. the maximum number of backedges on an acyclic program path.

Recently, Dhamdhere and Khedker [5, 14] generalized this result towards bidirectional problems. However, a major drawback of their setting is that it is pinned to round-robin iterations. Unfortunately, such a schedule does not fit well to situations where information is side-propagated along critical edges. In this light, it is not astonishing that their results on the convergence speed of bidirectional bit-vector analyses are quite disappointing. They replace the *depth* d of a flow graph by its *width* w , which is the number of non-conform edge traversals on an information flow path.³ Unfortunately, the width is not a structural property of the flow graph, but varies with the problem under consideration, and unlike d which is 0 for acyclic programs is not even bounded in this case. Actually, the notion of width does not match to the intuition associated with the name, as even “slim” programs may have a large width. An intuitive reason for this behaviour is given in Fig. 2a which shows a program fragment with a number of critical edges. Let us consider that information flow in this example follows the equation

$$Info(n) = \sum_{m \in pred(n)} Info(m) + \sum_{n' \in succ(m)} Info(n')$$

which means that the information at node n is set to true if the information at a predecessor or the information of any “sibling” of n is true.

We can easily see that the width of a flow graph with such a fragment directly depends on the number of critical edges, and therefore possibly grows linearly with the “length” of the program. It should be noted that such programs are by

³ Informatively, an information flow path is a sequence of backwards or forwards directed edges along which a change of information can be propagated. A forward traversal along a forward edge or a backward traversal along a backward edge are conform with a round-robin schedule proceeding (forwards) in reverse postorder. The other two kind of traversals are non-conform. Complemental notions apply to round-robin iterations proceeding in postorder.

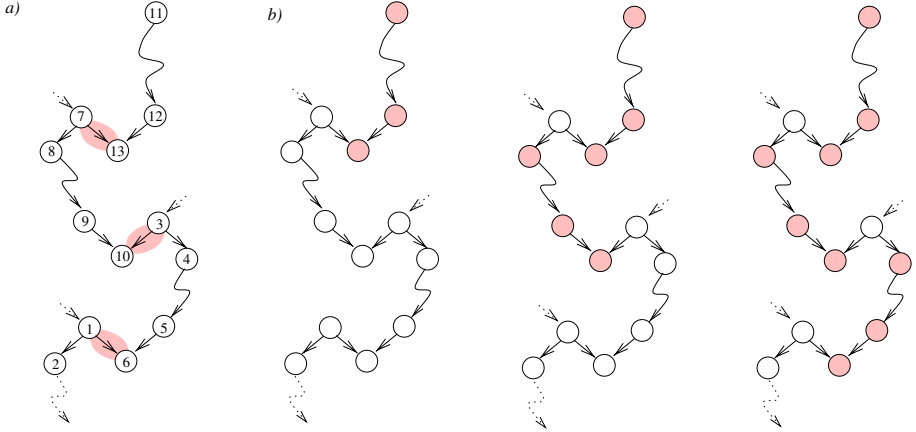


Fig. 2. a) Acyclic program path responsible for the width of a program with reverse postordering of the nodes b) Slow information propagation in round-robin iterations

no means pathological and thus the linear growth of the width is not unlikely for real-life programs. In fact, considering the reverse postorder of nodes as given in Fig. 2a the large width is actually reflected in a poor behaviour of a round-robin iteration. Fig. 2b shows how the information slowly propagates along the obvious “path” displayed in this example being stopped in each round-robin iteration at a non-conform (critical) edge⁴

Dhamdhere and Patil [6] proposed an elimination method for bidirectional problems that is as efficient as in the unidirectional case. However, it is restricted to a quite pathological class of problems, namely weakly bidirectional bit-vector problems and, as usual for elimination methods, is mainly designed for reducible control flow.

Finally, our hybrid approach shares with the hybrid iteration strategy of [11] that it mixes a round-robin schedule with exhaustive subiterations. However, their subiterations are within strongly connected components and their approach is solely designed to speed up unidirectional iterations.

2 Preliminaries

We consider programs in terms of *directed flow graphs* $= (N, E, \mathbf{s}, \mathbf{e})$ with node set N , edge set E and unique *start* and *end* nodes \mathbf{s} and \mathbf{e} , respectively. Nodes $n, m, \dots \in N$ represent (elementary) statements and are assumed to lie on a path from \mathbf{s} to \mathbf{e} . Finally, predecessors and successors of a node $n \in N$ are denoted by $\text{pred}(n)$ and $\text{succ}(n)$, respectively, and $\mathbf{P}[n, m]$ stands for the set of finite paths between node n and m .

⁴ Shaded circles indicate the flow of informations along the “path”.

Local Predicates. As usual our reasoning is based on an arbitrary but fixed expression φ that is the running object for code movement. With each node of the flow graph two local predicates are associated.

Comp(n): φ is computed at n , i. e. φ is part of the right-hand side expression associated with n .

Transp(n): n is transparent for φ , i. e. none of φ 's variables is modified at n .

Global Predicates. Based on these local predicates global program properties are specified. Usually, global predicates are associated with both entries and exits of nodes. In order to keep the presentation simple we assume that every node is split into an entry node and an empty exit node which inherit the set of predecessors and successors from the original node, respectively, and which are assumed to be connected by an edge leading from the entry node to the exit node. This step allows to restrict our reasoning to entry predicates. It should be noted, however, that this step is solely conceptual and does not eliminate any critical edge.

In this paper partial redundancy elimination (PRE), or code motion (CM) as a synonym, stands for program transformations that

1. insert some instances of initialisation statements $\mathbf{h}_\varphi := \varphi$ at program points, where \mathbf{h}_φ is a *temporary variable* that is exclusively assigned to φ and
2. replaces some original occurrences of φ by a usage of \mathbf{h}_φ .

In order to guarantee that the semantics of the argument program is preserved, we require that a code motion transformation must be *admissible*. Intuitively, this means that every insertion of a computation is *safe*, i. e. on no program path the computation of a new value is introduced at initialization sites, and that every substitution of an original occurrence of φ by \mathbf{h}_φ is *correct*, i. e. \mathbf{h}_φ always represents the same value as φ at use sites. This requires that \mathbf{h}_φ is properly initialized on every program path leading to some use site in a way such that no modification occurs afterwards.⁵

3 Code Motion in the Absence of Critical Edges

Before presenting our new approach to PRE in the presence of critical edges we shall first briefly recall the basic steps of lazy code motion [15] as a typical representative of an algorithm that relies on the absence of critical edges. Lazy code motion was the first algorithm for partial redundancy elimination that succeeded in removing partial redundancies as good as possible, while avoiding any unnecessary register pressure. This was mainly achieved by a rigorous redesign of Morel's and Renvoise's algorithm. Starting from a specification oriented view the key points was a hierarchical separation between the primary and the secondary concern of partial redundancy elimination, namely to minimize the number of

⁵ For a formal definition see [16].

computations and to avoid unnecessary register pressure, respectively. This hierarchical organization is reflected in a two-step design of the algorithm: lazy code motion rests on busy code motion. In following we briefly summarize the details of these transformations.

3.1 Busy Code Motion

Busy code motion (BCM) [15,16,9] places initializations *as early as possible* while replacing all original occurrences of φ . This is achieved by determining the *earliest* program points, where an initialization is *safe*. Technically, the range of safe program points can be determined by separately computing down-safe and up-safe program points. Both are given through the greatest solutions of two uni-directional data flow analyses, respectively.⁶

$$\begin{aligned}
 DnSafe(n) &= (n \neq \mathbf{e}) \cdot (Comp(n) + Transp(n) \cdot \prod_{m \in succ(n)} DnSafe(m)) \\
 UpSafe(n) &= (n \neq \mathbf{s}) \cdot Transp(m) \cdot \prod_{m \in pred(n)} (Comp(m) + UpSafe(m)) \\
 Safe(n) &\stackrel{\text{def}}{=} UpSafe(n) + DnSafe(n) \\
 Earliest(n) &\stackrel{\text{def}}{=} Safe(n) \cdot ((n = \mathbf{s}) + \sum_{m \in pred(n)} \overline{Safe(m)})
 \end{aligned}$$

Despite of its surprising simplicity, BCM already reaches *computational optimality*, i. e. programs resulting from this transformation have at most as many φ -occurrences on every path from `sto` `eas` any other result of an admissible code motion transformation (cp. [15,16]).

3.2 Lazy Code Motion

In addition to BCM, lazy code motion (LCM) takes the lifetimes of temporaries into account. This is accomplished by placing initialisations *as late as possible* but *as early as necessary*, where the latter requirement means “necessary in order to reach computational optimality”. Technically, this is achieved by determining the *latest* program points where a BCM-initialisation might be *delayed* to, which leads to one additional uni-directional data flow analysis.⁷

⁶ As common “.”, “+” and overlining stand for logical conjunction, disjunction and negation, respectively.

⁷ In [15,16] an additional analysis is employed determining *isolated* program points, i. e. program points where initialisations are only used immediately afterwards. This aspects, however, can independently be treated by means of a postprocess. For the sake of simplicity we skip the isolation analysis in this paper.

$$\begin{aligned}
 Delayed(n) &= Earliest(n) + (n \neq s) \cdot \prod_{m \in pred(n)} Delayed(m) \cdot \overline{Comp(m)} \\
 Latest(n) &\stackrel{\text{def}}{=} Delayed(n) \cdot (Comp(n) + \sum_{m \in succ(n)} \overline{Delayed(m)})
 \end{aligned}$$

LCM is computationally optimal as well as *lifetime optimal*, i.e. the temporary associated with φ has a lifetime range that is included in the lifetime range of any other program resulting from a computationally optimal code motion transformation (cp. [15,16]). The *lifetime range* of a temporary comprises all nodes whose exits occur in between an initialisation site and a use site such that no other initialisations are situated in between.⁸

4 Code Motion in the Presence of Critical Edges

In this section our new approach to PRE in the presence of critical edges is elaborated in full details. First we shall investigate the principal differences to the setting presented in Sect. 3.2. As opposed to flow graphs without critical edges there are usually no computationally optimal representatives. In fact, Fig. 3 shows two admissible, but computationally incomparable transformations that cannot be improved any further. The first one is simply given by the identical transformation of the program in Fig. 3a, the result of the second one is displayed in Fig. 3b. Each of the resulting programs has exactly one computation on the path that is emphasised in the dark shade of grey, while having two computations on the path being emphasised in the light shade of grey, respectively. Thus there is no computationally optimal code motion transformation with respect to the original program in Fig. 3a.

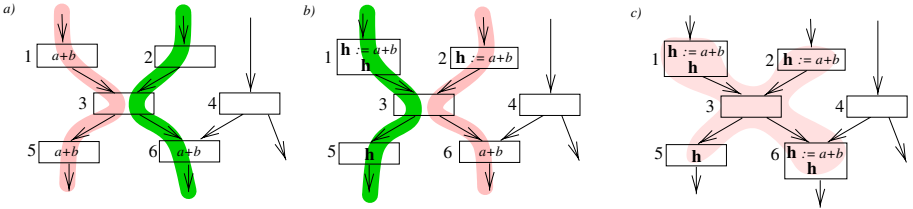


Fig. 3. a & b) Incomparable admissible program transformations c) Program degradation through a naive adaption of busy expression motion

⁸ In [20] we show that this optimality result is only adequate for flat universes of expressions. If both composite expressions and their subexpressions are moved, then the notion of lifetime optimality changes and a significantly more sophisticated technique has to be applied. Nonetheless, LCM still provides a basic ingredient of this approach.

This problem can be overcome by restricting the range of program transformations to those that are *profitable*, which means those that actually improve their argument programs. Note that this requirement excludes Fig. 3b as a reasonable code motion transformation. Obviously, profitability does not provide a further restriction for flow graphs without critical edges, where computationally optimal code motion transformations are granted to exist. In the presence of critical edges, however, this additional constraint is necessary in order to yield computationally optimal results at all.

4.1 Busy Code Motion

In this section we will develop a counterpart to BCM in the presence of critical edges. After briefly sketching the difficulties that prohibit a straightforward adaption of the uncritical solution, a correct approach is systematically developed from a specification that incorporates the special role of critical edges.

Unfortunately, BCM as presented in Sect. 3.1 cannot straightforwardly be applied to flow graphs with critical edges. This is because such a naive adaption may include non-profitable transformations as it is illustrated in Fig. 3c, where the marked range of down-safe program points would yield earliest initialisation points at nodes 1, 2 and 6⁹

Homogeneous Propagation of Down-Safety. The key for a useful critical variant of BCM is to impose an additional homogeneity requirement on down-safety that ensures that the information propagates either to all or to none of its predecessors, which grants that earliest program points become a proper upper borderline of the region of safe program points. In fact, in the absence of critical edges down-safety has the following homogeneity property:

$$\forall n \in N. DnSafe(n) \Rightarrow (\forall m \in pred(n). Safe(m) \vee \forall m \in pred(n). \neg DnSafe(m))$$

Note that the first term of the disjunction uses safety rather than down-safety, since propagation of down-safety needs not to be considered for predecessors that are up-safe anyhow¹⁰. Now this property has to be forced explicitly. For instance, in Fig. 3c node 6 as well as node 3 are down-safe, while node 4 is not. Therefore, let us consider the following notion of *homogeneous down-safety*:

Definition 1 (Homogeneous Down-Safety). A predicate *HDnSafe* on the nodes of *N* is a homogeneous down-safety predicate iff for any *n* ∈ *N*

1. *HDnSafe* is conform with down-safety:

$$HDnSafe(n) \Rightarrow (n \neq e) \wedge (Comp(n) \vee Transp(n) \wedge \forall m \in succ(n). HDnSafe(m))$$

⁹ Modifying this example by removing the computation of $a + b$ from node 1, would even result in a transformation that does not improve any path while strictly impairing some.

¹⁰ In the absence of critical edges this makes no difference to $\forall m \in pred(n). DnSafe(m)$.

2. *HDnSafe* is homogeneous:

$$HDnSafe(n) \Rightarrow (\forall m \in pred(n). (HDnSafe(m) \vee UpSafe(m)) \vee \forall m \in pred(n). \neg HDnSafe(m))$$

Obviously, homogeneous down-safety predicates are closed under “union”^[11]. Thus there exists a unique largest homogeneous down-safety predicate $DnSafe_{Hom}$, which gives rise to a homogeneous version of safety, too:

$$\forall n \in N. Safe_{Hom}(n) \stackrel{\text{def}}{=} DnSafe_{Hom}(n) \vee UpSafe(n)$$

It should be noted that this definition is developed from a pure specification oriented reasoning and can be seen as a first rigorous characterization of down-safety in the presence of critical edges: down safety is described by a backward directed data flow problem which is restricted by additional homogeneity constraints. This is in contrast to other algorithms, where bidirectional equation systems are postulated in an ad-hoc fashion without any separation of their functional components.

Earliest program points are defined as in the uncritical case, but with the difference of using the homogeneous version of down-safety in place of the usual one.

$$Earliest_{Hom}(n) \stackrel{\text{def}}{=} DnSafe_{Hom}(n) \cdot ((n \neq s) + \sum_{m \in pred(n)} \overline{Safe_{Hom}(m)})$$

The earliest program points serve as insertion points of BCM for flow graphs with critical edges (CBCM). With a similar argumentation as for BCM is easy to prove that CBCM is indeed *computationally optimal*, however, only relatively to the profitable transformations.

Computing CBCM: The Data Flow Analyses. In this part we present how the specifying solution of CBCM can be translated into appropriate data flow analyses determining the range of homogeneously safe program points. We will discuss three alternative approaches: (1) A “classical” one via bidirectional analyses, (2) a new non-standard approach that transforms the problem into one with purely unidirectional equations and (3) a hybrid approach that separates backwards flow from side propagation.

The Bidirectional Approach The specification of Definition [1] can straightforwardly be transferred into a bidirectional equation system for down-system.

¹¹ This means the predicate defined by the pointwise conjunction of the predicate values.

$$\begin{aligned}
UpSafe(n) &= (n \neq \mathbf{s}) \cdot Transp(m) \cdot \prod_{m \in pred(n)} (Comp(m) + UpSafe(m)) \\
DnSafe_{Hom}(n) &= (n \neq \mathbf{e}) \cdot (Comp(n) + Transp(n) \cdot \\
&\quad \prod_{m \in succ(n)} DnSafe_{Hom}(m) \cdot \prod_{n' \in pred(m)} (UpSafe(n') + DnSafe_{Hom}(n'))) \\
Safe_{Hom}(n) &\stackrel{\text{def}}{=} UpSafe(n) + DnSafe_{Hom}(n)
\end{aligned}$$

Unfortunately, the above bidirectional data flow problem shares the problems sketched in Fig. 2 when subjected to a round-robin iteration strategy. In fact, violation of homogeneous safety follows exactly the same definition pattern as *Info* does in this example.¹² Hence slow propagation of down-safety would be also apparent in CBCM.

The Unidirectional Approach It is easy to see that in the bidirectional equation system there is no “true” forward propagation of down-safety information as the scope of the term $\prod_{n' \in pred(m)} (UpSafe(n') + DnSafe_{Hom}(n'))$ is restricted in its context. Rather this can be seen as a “side propagation” of down-safety information along zig-zag paths. For a technical description let us define the set of *zig-zag successors* $zsucc(n)$ of a node $n \in N$ as the smallest set of nodes satisfying (see Fig. 4 for illustration):

1. $succ(n) \subseteq zsucc(n)$
2. $\forall m \in zsucc(n). succ(pred(m)) \subseteq zsucc(n)$

In our example zig-zag propagation of non-down-safety is further stopped at nodes where up-safety can be established. Hence we introduce a parameterized notion of $zsucc(n)$ which is defined for $M \subseteq N$ by:

1. $succ(n) \subseteq zsucc_M(n)$
2. $\forall m \in zsucc_M(n). succ(pred(m) \setminus M) \subseteq zsucc_M(n)$

With $XUS \stackrel{\text{def}}{=} \{m \in N \mid UpSafe(m)\}$ the equation for down-safety can be rewritten as:

$$DnSafe_{Hom}(n) = n \neq \mathbf{e} \cdot (Comp(n) + Transp(n) \cdot \prod_{m \in zsucc_{XUS}(n)} DnSafe_{Hom}(m))$$

Note that this equation system can be seen as a unidirectional one that operates on a flow graph that is enriched by shortcut edges drawn between nodes and their zig-zag successors (see Fig. 4b).

¹² Actually, here non-down-safety is propagated in a dual fashion.

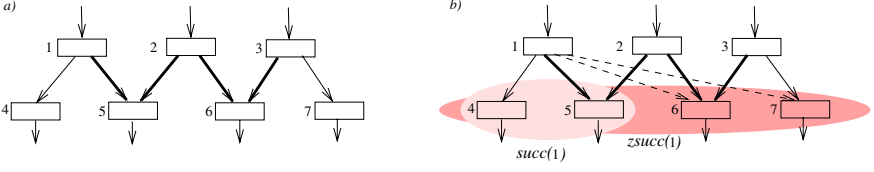


Fig. 4. (a) Program fragment with a nest of critical edges (b) Zig-zag successors and virtual shortcut edges of node 1

However, we do not actually recommend to perform such a transformation, as this would require to introduce an unnecessarily large number of additional edges. For a zig-zag chain of k critical edges as shown in Fig. 4a the number of shortcut edges is of order k^2 . Although, long zig-zag chains of critical edges can be expected to be rare in practice, we will show that information propagation can be organized without such blow-up in the number of edges. However, the important contribution of the unidirectional approach is that it provides the first meet over all paths characterization of PRE in the presence of critical edges.¹³

The Hybrid Approach: The hybrid approach rather addresses the organization of the iteration process than the equation system itself. As we have learned, bidirectional problems like our formulation of homogeneous down-safety do not fit together with a round-robin schedule based upon postorder traversals. The hybrid approach modifies the conventional round-robin schedule by integrating zig-zag propagation of information. This is achieved by clustering the nodes in a flow graph in a way such that side propagation of information can take benefit of much potential for simultaneous work. The overall schedule of the approach can be sketched as follows:

Preprocess: Collapsing of nodes according to side flow of information

Outer Schedule: Process the collapsed nodes in postorder until stabilization is reached performing an

Inner Schedule

1. For each node within the collapsed one perform information propagation along its outgoing uncritical edges
2. Perform exhaustive information propagation along the outgoing critical edges within the collapsed node

In the following we will go into the details of this process.

- The preprocess: Clustering of nodes groups together nodes of N according to the following equivalence relation:

$$n \equiv m \stackrel{\text{def}}{\Leftrightarrow} zsucc(n) = zsucc(m)$$

¹³ Actually, only the notion of paths has to be extended towards paths across shortcut edges.

It should be noted that G can be decomposed into its equivalence classes easily by tracing zig-zag paths of critical edges originating at an unprocessed node. For instance, starting with node 1 in Fig. 4a we obtain the equivalence class $\{1, 2, 3\}$ by following the critical edges. Clearly, this process can be managed in order $\mathcal{O}(e)$ where e denotes the number of edges in E . All nodes of an equivalence class are collapsed into a single node that inherits all incoming and outgoing edges of its members (see Fig. 5 for illustration).

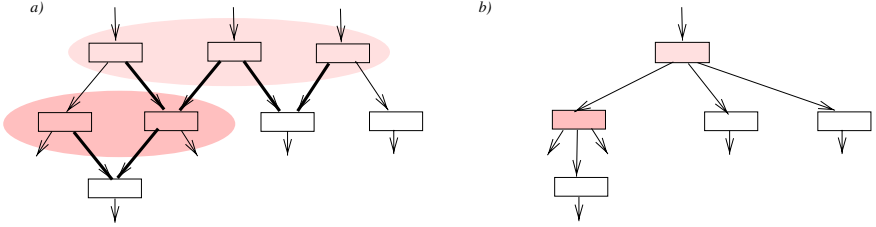


Fig. 5. (a) Equivalent nodes (b) Collapsing equivalent nodes

- The outer schedule: The flow graph G' that results from the collapsing pre-process is used in order to determine the round-robin schedule which drives information backwards. It should be noted that the depth of G' may differ from the depth of the original flow graph G in both directions: the depth may increase or decrease by collapsing. This is illustrated in Fig. 6. While collapsing nodes in Part a) decreases the depth, since the indicated path is no longer acyclic, collapsing in Part b) allows to construct a longer acyclic path as indicated by the dashed line connecting two independent acyclic paths.

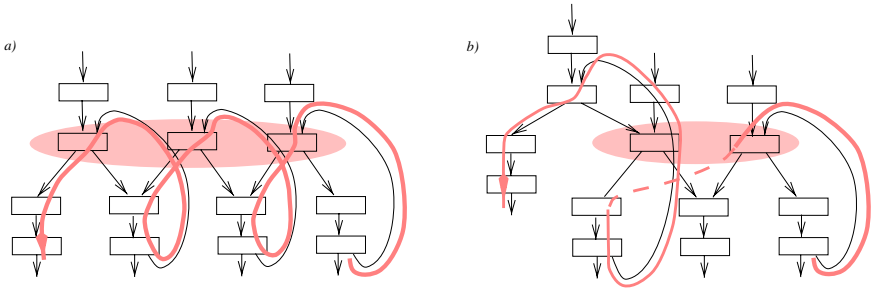


Fig. 6. (a) Decrease of depth due to collapsing of nodes (b) Increase of depth due to collapsing of nodes

- The inner schedule: The first step of the inner schedule is quite trivial. Considering a node $n \in N$ within the collapsed node under consideration and an

uncritical edge $(n, m) \in E$ the value of $DnSafe_{\text{Hom}}(n)$ is changed to false if and only if $Transp(n) \cdot DnSafe_{\text{Hom}}(m)$ holds.¹⁴

The second step of the inner schedule is the central innovation in our approach and has to be elaborated with some care. Within any collapsed node side propagation of down-safety information along critical edges is done exhaustively by using a subiteration process. Information propagation here means that for nodes n, m in the collapsed node under consideration with $m \in \text{pred}(\text{succ}(n))$ the value of $DnSafe_{\text{Hom}}(m)$ is changed to false if and only if $DnSafe_{\text{Hom}}(n) \cdot UpSafe(n)$ holds. The crucial point, however, is to organize the information flow along the critical edges. The situation is easy if the zig-zag paths are acyclically shaped as displayed in Fig. 7a or Fig. 7b. In this case the equivalence class can be represented as a tree, which can already be built while preprocessing this class. Following the topological order of the tree, information can be propagated completely by a bottom-up traversal (from the leaves to the root) followed by a top-down traversal (from the root to the leaves).

Unfortunately, in general there may be cycles of critical edges as shown in Fig. 7c and Fig. 7d. Hence a problem of the same difficulty as in the backward propagation of information shows up in the side-propagation step. However, separating both problems is useful as we expect nested cycles of critical edges to be a phenomenon that is extremely rare in practice. Nonetheless, to cope with them is quite straightforward. As in the acyclic case, the equivalence class can be represented as a tree with some additional non-tree edges establishing cycles. The only difference to the non-cyclic case is that the tree traversals have to be iterated more than once until the process gets stable. To estimate the number of traversal we borrow the arguments from conventional unidirectional analysis. Denoting the non-tree edges within the tree-like representation of an equivalence class as *critical backedges* the number of iterations is bound by d_c , where d_c is the maximum number of critical back edges along an acyclic path in any component representation.

Complexity of the Hybrid Approach: All together the iteration of homogeneous down-safety in the hybrid approach requires to apply the outer schedule until stabilization. Since the inner schedule propagates the information completely within each collapsed node the overall effort can be estimated by

$$(d' + 2)(e_u + 2(d_c + 2)e_c)$$

bit-vector steps, where e_u and e_c denote the number of uncritical and critical edges, respectively, d' is the depth of the collapsed flow graph G' and d_c the critical depth as defined before.

It is commonly argued that the depth of a flow graph is a reasonably small constant in practice. We already discussed that d_c is at least as likely to be a small constant, too. Hence the algorithm is expected to behave linear in e for real-life programs. In particular, we succeed in giving the first linear worst-case estimation for acyclic programs as in our introductory example of Fig. 2.

¹⁴ Note that there are no uncritical edges directly connecting different nodes of an equivalence class.

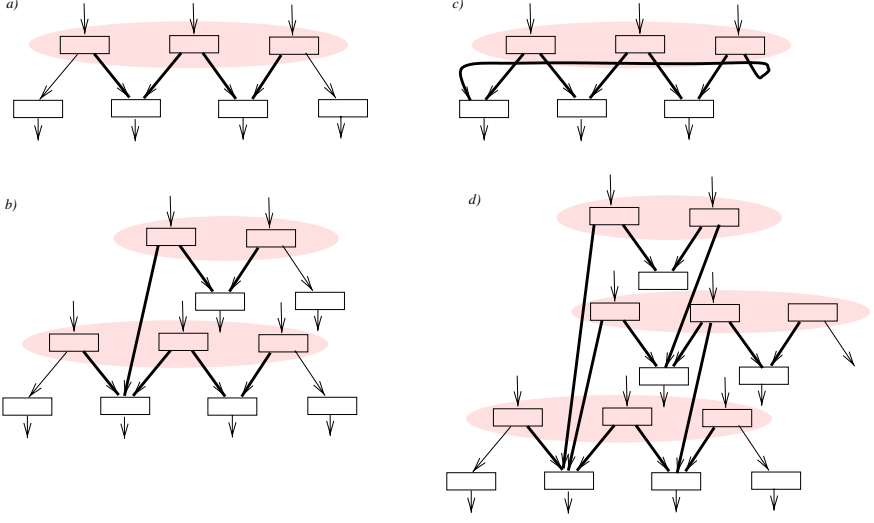


Fig. 7. Shapes of equivalent nodes: (a) chain of critical edges, (b) tree of critical edges, (c) cycle of critical edges and (d) structure with nested cycles of critical edges

4.2 Lazy Code Motion

Similar to the situation in Sect. 4.1 also the relevant analyses of LCM as defined in Sect. 3.2 cannot naively be adapted to flow graphs with critical edges. Again the reason for this behaviour lies in a homogeneity defect, but now with respect to delayability. In fact, for flow graphs without critical edges we have

$$Delayed(n) \Rightarrow (\forall m \in succ(n). Delayed(m) \vee \forall m \in succ(n). \neg Delayed(m))$$

This property may now be violated. Hence one has to force homogeneity explicitly in order to yield an appropriate critical variant of lazy code motion. Therefore, let us consider the following notion of *homogeneous delayability*.

Definition 2 (Homogeneous Delayability). A predicate $HDelayed$ on N is a homogeneous delayability predicate iff for any $n \in N$

1. $HDelayed(n)$ is conform with delayability:

$$HDelayed(n) \Rightarrow \text{Earliest}_{\text{Hom}}(n) \vee ((n \neq s) \wedge \forall m \in pred(n). HDelayed(m) \wedge \neg Comp(m))$$

2. $HDelayed(n)$ is homogeneous:

$$Delayed(n) \Rightarrow (\forall m \in succ(n). HDelayed(m) \vee \forall m \in succ(n). \neg HDelayed(m))$$

Obviously, homogeneous delayability predicates are closed under “union”. Thus there exists a unique largest homogeneous delayability predicate $Delayed_{\text{Hom}}$. This gives rise to a new version of latestness characterizing the insertion points of lazy code motion for flow graphs with critical edges (CLCM).

$$Latest_{\text{Hom}}(n) \stackrel{\text{def}}{\Leftrightarrow} Delayed_{\text{Hom}}(n) \wedge (Comp(n) \vee \exists m \in succ(n). \neg Delayed_{\text{Hom}}(m))$$

Using the same definitions for lifetime optimality as in Sect. 3.2 we succeed in proving lifetime optimality of CLCM.

Computing CLCM. In analogy to Sect. 11 the delayability property can either be coded into a bidirectional equation system or, more interestingly, again be expressed using a unidirectional formulation:

$$\begin{aligned} \text{Delayed}_{\text{Hom}}(n) = & \text{Earliest}_{\text{Hom}}(n) \vee \\ & ((n \neq \mathbf{s}) \wedge \forall m \in \text{zpred}(n). \text{Delayed}_{\text{Hom}}(m) \wedge \neg \text{Comp}(m)) \end{aligned}$$

This definition is based on zig-zag predecessor, which are defined completely along the lines of zig-zag successors. However, in contrast to down-safety *zpred* needs not to be parameterized this time. Using this characterization the same techniques for hybrid iteration can be used as in Sect. 11

5 Conclusion

We presented an adaption of lazy code motion to flow graphs with critical edges as a model how to cope with bidirectional dependencies in code motion. On the conceptual level we isolated homogeneity requirements as the source for bidirectional dependencies. This led to a new hybrid iteration strategy which is almost as fast as its unidirectional counterparts. This dramatically improves all known estimations for bidirectional bit-vector methods. Nonetheless, we still recommended to eliminate critical edges as far as possible, since critical edges are also responsible for problems of a different flavour [19]. However, any implementation of code motion that has to cope with critical edges will definitely benefit from the ideas presented in this paper.

References

1. D. M. Dhamdhere. A fast algorithm for code movement optimization. *ACM SIGPLAN Notices*, 23(10):172 – 180, 1988.
2. D. M. Dhamdhere. A new algorithm for composite hoisting and strength reduction optimisation (+ Corrigendum). *International Journal of Computer Mathematics*, 27:1 – 14 (+ 31 – 32), 1989.
3. D. M. Dhamdhere. A usually linear algorithm for register assignment using edge placement of load and store instructions. *Journal of Computer Languages*, 15(2):83 – 94, 1990.
4. D. M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291 – 294, 1991. Technical Correspondence.
5. D. M. Dhamdhere and U. P. Khedker. Complexity of bidirectional data flow analysis. In *Conf. Record of the 20th ACM Symposium on the Principles of Programming Languages*, pages 397–409, Charleston, SC, January 1993.

6. D. M. Dhamdhere and H. Patil. An elimination algorithm for bidirectional data flow problems using edge placement. *ACM Transactions on Programming Languages and Systems*, 15(2):312 – 336, April 1993.
7. D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'92*, volume 27,7 of *ACM SIGPLAN Notices*, pages 212 – 223, San Francisco, CA, June 1992.
8. K.-H. Drechsler and M. P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635 – 640, 1988. Technical Correspondence.
9. K.-H. Drechsler and M. P. Stadel. A variation of Knoop, Rüthing and Steffen's lazy code motion. *ACM SIGPLAN Notices*, 28(5):29 – 38, 1993.
10. M. S. Hecht and J. D. Ullman. A simple algorithm for global data flow analysis problems. *SIAM Journal on Computing*, 4(4):519 – 532, 1977.
11. S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for data flow analysis. *Acta Informatica*, 24:679 – 694, 1987.
12. S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization – part I. *International Journal of Computer Mathematics*, 11:21 – 41, 1982.
13. S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization – part II. *International Journal of Computer Mathematics*, 11:111 – 126, 1982.
14. U. P. Khedker and D. M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM Transactions on Programming Languages and Systems*, 16(5):1472 – 1511, September 1994.
15. J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'92*, volume 27,7 of *ACM SIGPLAN Notices*, pages 224 – 234, San Francisco, CA, June 1992.
16. J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.
17. P. M. Masticola, T. J. Marlowe, and B. G. Ryder. Lattice frameworks for multi-source and bidirectional data flow problems. *ACM Transactions on Programming Languages and Systems*, 17(5):777 – 802, 1995.
18. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96 – 103, 1979.
19. O. Rüthing. *Interacting Code Motion Transformations. Their Impact and their complexity*. PhD thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Germany, 1997. Available as <http://sunshine.cs.uni-dortmund.de/~ruething/diss.ps.gz>
20. O. Rüthing. Optimal code motion in the presence of large expressions. In *Proc. International Conference on Computer Languages (ICCL'98)*, Chicago, IL., 1998. IEEE.

On a New Method for Dataflow Analysis of Java Virtual Machine Subroutines

Masami Hagiya and Akihiko Tozawa

Department of Information Science,
Graduate School of Science,
University of Tokyo
{hagiya,miles}@is.s.u-tokyo.ac.jp

Abstract. The bytecode verifier of the Java Virtual Machine, which statically checks the type safety of Java bytecode, is the basis of the security model of Java and guarantees the safety of mobile code sent from an untrusted remote host. However, the type system for Java bytecode has some technical problems, one of which is in the handling of subroutines. Based on the work of Stata and Abadi and that of Qian, this paper presents yet another type system for Java Virtual Machine subroutines. Our type system includes types of the form `last(x)`. A value whose type is `last(x)` is the same as that of the x -th variable of the caller of the subroutine. In addition, we represent the type of a return address by the form `return(n)`, which means returning to the n -th outer caller. By virtue of these types, we can analyze instructions purely in terms of type, and as a result the correctness proof of bytecode verification becomes extremely simple. Moreover, for some programs, our method is more powerful than existing ones. In particular, our method has no restrictions on the number of entries and exits of a subroutine.

1 Introduction

One contribution of Java is its bytecode verifier, which statically checks the type safety of bytecode for the JVM (Java Virtual Machine) prior to execution. Thanks to the bytecode verifier, bytecode sent from an untrusted remote host can be executed without the danger of causing type errors and destroying the entire security model of Java, even when the source code is not available. Verifying the type safety of bytecode (or native code) seems to be a new research area that is not only of technical interest but also of practical importance, due to the availability of remote binary code in web browsers and other applications.

Much effort has been put into guaranteeing the security of Java programs, including the type safety of bytecode:

- Security model for Java applets: The security model for Java applets is said to consist of three *prongs*: the bytecode verifier, the applet class loader and the security manager [9]. In this model, the bytecode verifier plays the most fundamental role, on which the other two prongs are based. If the bytecode verifier is cheated, the other two also become ineffective.

- Type safety of source code: The type safety of Java programs has been proved, either formally (using a theorem proving assistant), or rigorously (but not formally) [3,4,7,12]. This means that a program that has passed static type checking is guaranteed to cause no type errors while it is running.
- Safety of class loading: Java allows classes to be loaded lazily, i.e., only when classes are actually accessed. In order to support various loading disciplines, Java allows programmers to define their own class loaders. This has opened one of the security holes in Java [14]. To avoid such a security hole, Dean formalized part of the class loader functionality and formally proved its correctness using PVS [2]. Goldberg’s main concern is also class loading, though bytecode verification is addressed [5].
- Type safety of bytecode: The bytecode verifier statically checks the type safety of Java bytecode. If the bytecode verifier accepts incorrectly typed bytecode, it will break the entire security model of Java. It guarantees that no type error occurs at each instruction by performing dataflow analysis on the bytecode.

Besides the researches into different aspects of security mentioned above, there are also some on-going projects that are developing more secure network programming environments [7,15].

This paper concerns bytecode verification. Since this is the basis of the entire security model of Java, it is desirable to rigorously prove that any bytecode program that has passed bytecode verification will never cause a runtime type error.

In order to be able to show the correctness of the bytecode verifier, one has to begin by formally specifying the operational semantics of the virtual machine (e.g., [1]), and then give the formal specification of the bytecode verifier, based on its informal specification written in English [8].

Qian rigorously defined the operational semantics of a subset of the JVM and formulated the bytecode verifier as a type system [13]. He then succeeded in proving the correctness of the bytecode verifier, though not completely.

Bytecode verification of the JVM has some technical challenges. One is that of handling object initialization, as objects created but not yet initialized may open a security hole. In Qian’s work, much attention is paid to the handling of object initialization.

Another is that of handling the polymorphism of subroutines. This paper addresses this issue. Inside a JVM subroutine, which is the result of compiling a **finally** clause in a **try** statement, local variables may have values of different types, depending on the caller of the subroutine. This is a kind of polymorphism.

To investigate how to analyze JVM subroutines, Stata and Abadi defined a type system for a small subset of the JVM and proved its correctness with respect to the operational semantics of the subset [16]. Qian’s system is similar to that of Stata and Abadi in its handling of subroutines [13]. Both systems faithfully follow the specification of the bytecode verifier, and make use of information as to which variables are accessed or modified in a subroutine. Those variables that are not accessed or modified are simply ignored during analysis of the subroutine.

This paper takes a different approach. We introduce types of the form `last(x)`. A value whose type is `last(x)` is the same as that of the x -th variable of the caller of the subroutine. In addition, we represent the type of a return address by the form `return(n)`, which means returning to the n -th outer caller.

Our approach has the following advantages.

- By virtue of the `last` and `return` types, we can analyze instructions purely in terms of types. As a result, the proof of the correctness of bytecode verification becomes extremely simple, and we do not need a separate analysis on variable access or modification.
- For some programs (unfortunately, not those produced by the Java compiler), our method is more powerful than existing ones. In particular, our method has no restrictions on the number of entries and exits of a subroutine. Stata and Abadi enforce a stack-like behavior on subroutine calls with their analysis, which does not account for out-of-order returns from subroutines, though they are actually produced by the Java compiler.

Due to these advantages, we hope that our method can be modified and applied to the analysis of other kinds of bytecode or native code [10, 11].

This paper is organized as follows: In the next section, we explain JVM subroutines in more detail and our approach to bytecode verification. In Sect. 3, a subset of the JVM, similar to that of Stata and Abadi, is defined. In Sect. 4, our method for analysis of bytecode is described and its correctness is shown. In Sect. 5, issues of implementation are briefly discussed. Section 6 offers concluding remarks.

2 Analysis of Subroutines

The JVM is a classical virtual machine consisting of

- a program counter,
- an array for storing the values of local variables,
- a stack for placing arguments and results of operators, called the *operand stack*,
- a heap for storing method code and object bodies, and
- a stack for frames, each of which is allocated for each invocation of a method and consists of the program counter, the local variables, and the operand stack.

To allow for checking the type safety of bytecode, it prepares different instructions for the same operation depending on the type of the operands. For example, it has the instruction `istore` for storing integers and the instruction `fstore` for storing floating-point numbers.

JVM subroutines are used mainly for compiling the `finally` clauses of `try` statements of Java. Notice that subroutine calls are completely different from method calls. A subroutine is locally defined inside a method and, unlike a method, is not allowed to call itself recursively.

In this paper, we define a virtual machine based on JVMLO, which was formulated by State and Abadi for investigating how to analyze JVM subroutines. Subroutines are called by instructions of the following form.

$$\text{jsr}(L)$$

An instruction of this form pushes the address of its next instruction (i.e., the return address) onto the operand stack and jumps to the subroutine L . Subroutines are usually defined as follows.

$$\begin{array}{c} L : \text{store}(x) \\ \vdots \\ \text{ret}(x) \end{array}$$

$\text{store}(x)$ pops a value from the operand stack and stores it in the x -th local variable. $\text{ret}(x)$ is an instruction for jumping to the address stored in the x -th local variable. Note that, in contrast to the JVM, our virtual machine has only one instruction for the store operation.

Subroutines in the JVM have made bytecode verification more difficult for the following reasons.

- The return address of $\text{ret}(x)$ can only be determined after the values of the local variables have been analyzed by the bytecode verifier. On the other hand, return addresses affect the control flow and the analysis of local variables.
- In some situations, a subroutine does not return to the immediate caller, but returns to an outer caller, such as the caller of the caller.
- Inside a subroutine, local variables may have values of different types, depending on the caller of the subroutine.

In this paper, we introduce types of the form $\text{last}(x)$ in order to address the last problem. A value having this type must have the same value as that of the x -th local variable in the caller of a subroutine.

As an example, let us consider the following program.

<code>const0</code>	<code>7 : store(0)</code>
<code>store(1)</code>	<code>load(1)</code>
<code>2 : jsr(7)</code>	<code>store(2)</code>
<code>constNULL</code>	<code>ret(0)</code>
<code>store(1)</code>	
<code>5 : jsr(7)</code>	
<code>halt</code>	

Subroutine 7 is called from two callers (2 and 5). The return address is stored in variable 0 (the 0-th local variable). The value of variable 1 is an integer when the subroutine is called from caller 2, and is an object pointer when called from caller 5. This is a typical case in which a subroutine is polymorphic. In this

program, the value of variable 1 is copied to variable 2. As the JVM has different store instructions depending on the type of the operand, the above program is impossible under the JVM. However, even if the JVM allowed the untyped store instruction, the bytecode verifier of the JVM would signal an error for the above program, because it always assigns a unique type for any variable accessed in a subroutine.

According to the specification of the JVM [8],

- For each instruction and each *jsr* needed to reach that instruction, a bit vector is maintained of all local variables accessed or modified since the execution of the *jsr* instruction.
- For any local variable for which the bit vector (constructed above) indicates that the subroutine has accessed or modified, use the type of the local variable at the time of the *ret*.
- For other local variables, use the type of the local variable before the *jsr* instruction.

The work of Stata and Abadi and that of Qian faithfully follow this specification. Local variables that are accessed or modified in each subroutine are recorded. Those variables that are not accessed or modified are simply ignored during the subsequent analysis of the subroutine.

The method proposed in this paper assigns a type of the form **last**(1) to variable 1 in subroutine 7. This means that it includes a value passed from variable 1 of the caller. This type is propagated through instructions in the subroutine. In particular, by the instruction **store**(2), the type of variable 2 becomes **last**(1). This information is then used when the control returns from the subroutine to the caller. In this way, the polymorphism of local variables in a subroutine is expressed by types of the form **last**(*x*).

In our method, return addresses have types of the form **return**(*n*). A type of the form **return**(*n*) means to return to the *n*-th outer caller. For example, the address returning to the immediate caller has type the **return**(1), while the address returning to the caller of the caller has type the **return**(2).

In Stata and Abadi’s work (and similarly in Qian’s work), return addresses have types of the form (**ret-from** *L*), where *L* is the address of the subroutine, from which the callers of the subroutines are obtained. In our analysis, the callers are obtained from the set of histories assigned to each instruction (cf. Sect. 4.3).

3 Virtual Machine

In this section we formalize a subset of the JVM, which resembles that of Stata and Abadi [16]. Differences are mainly for the examples by which we want to show the power of our framework.

3.1 Values

A value is a return address or an integer or an object pointer. We can easily add other kinds of values, such as that of floating point number. In the following formal treatment of the operational semantics of the virtual machine, a return address has the constructor **retaddr**, an integer the constructor **intval**, and an object pointer the constructor **objval**. They all take an integer as an argument.

3.2 Instructions

A bytecode program is a list of instructions. An instruction takes one of the following formats.

jsr (L)	(L : subroutine address)	ret (x)	(x : variable index)
load (x)	(x : variable index)	store (x)	(x : variable index)
const0		constNULL	
inc (x)	(x : variable index)		
if0 (L)	(L : branch address)	ifNULL (L)	(L : branch address)
halt			

Each mnemonic is considered as a constructor of instructions. Some of the mnemonics takes a nonnegative integer x or L as an operand.

3.3 Operational Semantics

The virtual machine consists of

- the program, which is a list of instructions and denoted by P ,
- the program counter, which is an index to P ,
- the local variables, where the list of values of the local variables is denoted by f , and
- the operand stack, denoted by s .

Let us use the notation $l[i]$ for extracting the i -th element of list l , where the first element of l has the index 0. The i -th instruction of the program P is denoted by $P[i]$. The value of the x -th local variable is denoted by $f[x]$. The p -th element of the operand stack s is denoted by $s[p]$, where $s[0]$ denotes the top element of s .

As in the work by Stata and Abadi, the operational semantics of the virtual machine is defined as a transition relation between triples of the form $\langle i, f, s \rangle$, where i is the program counter, i.e., the index to the program P , f the value list of the local variables, and s the operand stack. While the length of s may change during execution of the virtual machine, the length of f , i.e., the number of local variables is unchanged. The program P , of course, never changes during execution.

The transition relation is defined as follows.

- If $P[i] = \text{jsr}(L)$, then $\langle i, f, s \rangle \rightarrow \langle L, f, \text{retaddr}(i+1)::s \rangle$.
The return address $\text{retaddr}(i+1)$ is pushed onto the operand stack.
The operator $::$ is the *cons* operator for lists.
- If $P[i] = \text{ret}(x)$ and $f[x] = \text{retaddr}(j+1)$, then $\langle i, f, s \rangle \rightarrow \langle j+1, f, s \rangle$.
- If $P[i] = \text{load}(x)$, then $\langle i, f, s \rangle \rightarrow \langle i+1, f, f[x]::s \rangle$.
- If $P[i] = \text{store}(x)$, then $\langle i, f, v::s \rangle \rightarrow \langle i+1, f[x \mapsto v], s \rangle$.
The notation $f[x \mapsto v]$ means a list whose element is the same as that of f except for the x -th element, which is set to v .
- If $P[i] = \text{const0}$, then $\langle i, f, s \rangle \rightarrow \langle i+1, \text{intval}(0)::s \rangle$.
- If $P[i] = \text{constNULL}$, then $\langle i, f, s \rangle \rightarrow \langle i+1, \text{objval}(0)::s \rangle$.
- If $P[i] = \text{inc}(x)$ and $f[x] = \text{intval}(k)$, then
 $\langle i, f, s \rangle \rightarrow \langle i+1, f[x \mapsto \text{intval}(k+1)], s \rangle$.
- If $P[i] = \text{if0}(L)$, then $\langle i, f, \text{intval}(0)::s \rangle \rightarrow \langle L, f, s \rangle$.
If $P[i] = \text{if0}(L)$ and $k \neq 0$, then $\langle i, f, \text{intval}(k)::s \rangle \rightarrow \langle i+1, f, s \rangle$.
- If $P[i] = \text{ifNULL}(L)$, then $\langle i, f, \text{objval}(0)::s \rangle \rightarrow \langle L, f, s \rangle$.
If $P[i] = \text{ifNULL}(L)$ and $k \neq 0$, then $\langle i, f, \text{objval}(k)::s \rangle \rightarrow \langle i+1, f, s \rangle$.

The transition relation \rightarrow is considered as the least relation satisfying the above conditions.

The relation is defined so that when a type error occurs, no transition is defined. This means that to show the type safety of bytecode is to show that a transition sequence stops only at the **halt** instruction.

For proving the correctness of our bytecode analysis, we also need another version of the operational semantics that maintains invocation histories of subroutines. This semantics corresponds to the *structured dynamic semantics* of Stata and Abadi. The transition relation is now defined for quadruples of the form $\langle i, f, s, h \rangle$, where the last component h is an invocation history of subroutines. It is a list of addresses of callers of subroutines. This component is only changed by the **jsr** and **ret** instructions.

- If $P[i] = \text{jsr}(L)$, then $\langle i, f, s, h \rangle \rightarrow \langle L, f, \text{retaddr}(i+1)::s, i::h \rangle$.
Note that the address i of the caller of the subroutine is pushed onto the invocation history.
- If $P[i] = \text{ret}(x)$, $f[x] = \text{retaddr}(j+1)$ and $h = h'@[j]@h''$, where j does not appear in h' , then $\langle i, f, s, h \rangle \rightarrow \langle j+1, f, s, h'' \rangle$.
The operator $@$ is the *append* operator for lists.

For other instructions, the invocation histories before and after transition are the same.

As for the two transition relations, we immediately have the following proposition.

Proposition 1: If $\langle i, f, s, h \rangle \rightarrow \langle i', f', s', h' \rangle$, then $\langle i, f, s \rangle \rightarrow \langle i', f', s' \rangle$.

4 Analysis

4.1 Types

Types in our analysis are among the following syntactic entities:

\top, \perp	(top and bottom)	$\text{INT}, \text{OBJ}, \dots$	(basic types)
$\text{return}(n)$	(n : caller level)	$\text{last}(x)$	(x : variable index)

A type is \top , \perp , a basic type, a **return** type, or a **last** type. In this paper, we assume as basic types **INT**, the type of integers, and **OBJ**, the type of object pointers. It is easy to add other basic types, such as that of floating point numbers.

return types and **last** types are only meaningful inside a subroutine. A **return** type is the type of a return address. For positive integer n , $\text{return}(n)$ denotes the type of the address for returning to the n -th outer caller. For example, $\text{return}(1)$ denotes the type of the address for returning to the direct caller of the subroutine, and $\text{return}(2)$ the type of the address for returning to the caller of the caller.

A **last** type means that a value is passed from the caller of the subroutine. For nonnegative integer x , $\text{last}(x)$ denotes the type of a value that was stored in the x -th local variable of the caller. A value can have this type only when it is exactly the same as the value of the x -th local variable when the subroutine was called.

4.2 Order among Types

We define the order among types as follows.

$$\begin{array}{ll} \top > \text{INT} > \perp & \top > \text{OBJ} > \perp \\ \top > \text{return}(n) > \perp & \top > \text{last}(x) > \perp \end{array}$$

Since we do not distinguish object pointers by their classes in this paper, the order is *flat*, with \top and \perp as the top and bottom elements.

This order is extended to lists of types. For type lists \mathbf{t}_1 and \mathbf{t}_2 , $\mathbf{t}_1 > \mathbf{t}_2$ holds if and only if \mathbf{t}_1 and \mathbf{t}_2 are of the same length and $\mathbf{t}_1[i] > \mathbf{t}_2[i]$ holds for any i ranging over the indices for the lists.

4.3 Target of Analysis

The target of our bytecode analysis is to obtain the following pieces of information for the i -th instruction of the given program P .

$$F_i \quad S_i \quad H_i$$

F_i is a type list. $F_i[x]$ describes the type of $f[x]$, i.e., the value of the x -th local variable of the virtual machine. S_i is a also type list. Each element of S_i describes the type of the corresponding element of the operand stack of the virtual machine. Both F_i and S_i describe the types of the components of the virtual machine just before the i -th instruction is executed. H_i is a set of invocation histories for the i -th instruction.

F , S and H should follow a rule that is defined for each kind of $P[i]$. The rule says that certain conditions must be satisfied before and after the execution of $P[i]$.

Rule for jsr) If $h \in H_i$ and $P[i] = \text{jsr}(L)$, then the following conditions must be satisfied.

- $0 \leq L < |P|$.
- For each variable index y , either
 - $F_L[y] \geq \text{return}(n+1)$
(if $F_i[y] = \text{return}(n)$), and
 - $F_L[y] \geq F_i[y]$
(if $F_i[y]$ is neither **return** nor **last**)
- or
 - $F_L[y] \geq \text{last}(y)$
(even if $F_i[y]$ is **last**).
- $|S_L| = |S_i| + 1$, where $|l|$ denotes the length of list l .
- $S_L[0] \geq \text{return}(1)$.
- For each index p , where $0 \leq p < |S_i|$,
 - $S_i[p]$ is not **last**,
 - $S_L[p+1] \geq S_i[p]$
(if $S_i[p]$ is not **return**), and
 - $S_L[p+1] \geq \text{return}(n+1)$
(if $S_i[p] = \text{return}(n)$).
- i does not appear in h . (Recursion is not allowed.)
- $i::h \in H_L$.

Note that when $F_i[y]$ is not **last**, $F_L[y]$ cannot be determined uniquely. We must make a nondeterministic choice between $\text{return}(n+1)$ and $F_i[y]$. See Sect. 5 for more discussions on the implementation of the analysis.

The following figures show the two possibilities for typing local variables inside a subroutine. In this example, it is assumed that there is only one caller (2) of subroutine 7. The column $[F_i[0], F_i[1], F_i[2]]$ shows the types of local variables before each instruction is executed. At subroutine 7, it is set to $[\top, \text{INT}, \top]$ or $[1(0), 1(1), 1(2)]$. There are more possibilities. For example, one could also set it to $[\top, \top, \top]$, but this possibility is subsumed by the first.

i instruction	$[F_i[0], F_i[1], F_i[2]]$	S_i	H_i	$[F_i[0], F_i[1], F_i[2]]$	S_i	H_i
0 const0	$[\top, \top, \top]$	\emptyset	$\{\emptyset\}$	$[\top, \top, \top]$	\emptyset	$\{\emptyset\}$
1 store (1)	$[\top, \top, \top]$	$[\text{INT}]$	$\{\emptyset\}$	$[\top, \top, \top]$	$[\text{INT}]$	$\{\emptyset\}$
2 jsr (7)	$[\top, \text{INT}, \top]$	\emptyset	$\{\emptyset\}$	$[\top, \text{INT}, \top]$	\emptyset	$\{\emptyset\}$
3 constNULL	$[\top, \text{INT}, \text{INT}]$	\emptyset	$\{\emptyset\}$	$[\top, \text{INT}, \text{INT}]$	\emptyset	$\{\emptyset\}$
...						
7 store (0)	$[\top, \text{INT}, \top]$	$[\mathbf{r}(1)]$	$\{[2]\}$	$[1(0), 1(1), 1(2)]$	$[\mathbf{r}(1)]$	$\{[2]\}$
8 load (1)	$[\mathbf{r}(1), \text{INT}, \top]$	\emptyset	$\{[2]\}$	$[\mathbf{r}(1), 1(1), 1(2)]$	\emptyset	$\{[2]\}$
9 store (2)	$[\mathbf{r}(1), \text{INT}, \top]$	$[\text{INT}]$	$\{[2]\}$	$[\mathbf{r}(1), 1(1), 1(2)]$	$[1(1)]$	$\{[2]\}$
10 ret (0)	$[\mathbf{r}(1), \text{INT}, \text{INT}]$	\emptyset	$\{[2]\}$	$[\mathbf{r}(1), 1(1), 1(1)]$	\emptyset	$\{[2]\}$

Rule for ret) If $h \in H_i$ and $P[i] = \text{ret}(x)$, then the following conditions must be satisfied.

- $F_i[x] = \text{return}(n)$.

- $h = h' @ [j] @ h''$, where $|h'| = n - 1$.
- $0 \leq j + 1 < |P|$.
- For each variable index y ,
 $F_{j+1}[y] \geq \text{follow_last}(n, h, F_i[y])$.
- $S_{j+1} \geq \text{follow_last}(n, h, S_i)$.
- $h'' \in H_{j+1}$.

follow_last is a function for extracting the type of a variable in a caller of a sub-routine according to an invocation history. For nonnegative integer n , invocation history h and type t , *follow_last*(n, h, t) is defined as follows.

$$\begin{aligned}
 \text{follow_last}(0, h, t) &= t \\
 \text{follow_last}(n + 1, i :: h, \text{return}(m)) &= \\
 &\quad \text{if } m > n + 1 \text{ then } \text{return}(m - n - 1) \text{ else } \top \\
 \text{follow_last}(n + 1, i :: h, \text{last}(x)) &= \text{follow_last}(n, h, F_i[x]) \\
 \text{follow_last}(n + 1, i :: h, t) &= t \quad (\text{otherwise})
 \end{aligned}$$

follow_last is extended to type lists, i.e., *follow_last*(n, h, t) is also defined when t is a type list.

Rule for load) If $h \in H_i$ and $P[i] = \text{load}(x)$, then the following conditions must be satisfied.

- $0 \leq i + 1 < |P|$. $F_{i+1} \geq F_i$. $S_{i+1} \geq F_i[x] :: S_i$. $h \in H_{i+1}$.

Rule for store) If $h \in H_i$ and $P[i] = \text{store}(x)$, then the following conditions must be satisfied.

- $0 \leq i + 1 < |P|$. $S_i = t :: t$. $F_{i+1} \geq F_i[x \mapsto t]$. $S_{i+1} \geq t$. $h \in H_{i+1}$.

Rule for const0) If $h \in H_i$ and $P[i] = \text{const0}$, then the following conditions must be satisfied.

- $0 \leq i + 1 < |P|$. $F_{i+1} \geq F_i$. $S_{i+1} \geq \text{INT} :: S_i$. $h \in H_{i+1}$.

The rule for **constNULL** is similar.

Rule for inc) If $h \in H_i$ and $P[i] = \text{inc}(x)$, then the following conditions must be satisfied.

- $0 \leq i + 1 < |P|$. $F_i[x] = \text{INT}$. $F_{i+1} \geq F_i$. $S_{i+1} \geq S_i$. $h \in H_{i+1}$.

Rule for if0) If $h \in H_i$ and $P[i] = \text{if0}(L)$, then the following conditions must be satisfied.

- $0 \leq L < |P|$. $0 \leq i + 1 < |P|$.
- $S_i = \text{INT} :: t$. $F_L \geq F_i$. $F_{i+1} \geq F_i$. $S_L \geq t$. $S_{i+1} \geq t$.
- $h \in H_L$. $h \in H_{i+1}$.

The rule for **ifNULL** is similar.

Rule for halt) There is no rule for **halt**.

4.4 Correctness of Analysis

In order to state the correctness of our analysis, we first introduce the following relation.

$$\langle v, h \rangle : t$$

v is a value and h is an invocation history. t is a type. By $\langle v, h \rangle : t$, we mean that the value v belongs to the type t provided that v appears with the invocation history h . Following is the definition of this relation.

- $\langle v, h \rangle : \top$.
- $\langle \text{intval}(k), h \rangle : \text{INT}$.
- $\langle \text{objval}(k), h \rangle : \text{OBJ}$.
- If $h[n-1] = j$, then $\langle \text{retaddr}(j+1), h \rangle : \text{return}(n)$.
- If $\langle v, h \rangle : F_i[x]$, then $\langle v, i::h \rangle : \text{last}(x)$.

This definition is also inductive, i.e., $\langle v, h \rangle : t$ holds if and only if it can be derived only by the above rules.

We have two lemmas.

Lemma 1: If $\langle v, h \rangle : t$ and $t' \geq t$, then $\langle v, h \rangle : t'$.

Lemma 2: Let h' be a prefix of h of length n and h'' be its corresponding suffix, i.e., $h = h'@h''$ and $|h'| = n$. If $\langle v, h \rangle : t$, then $\langle v, h'' \rangle : \text{follow_last}(n, h, t)$.

We say that the quadruple $\langle i, f, s, h \rangle$ is sound with respect to $\langle F, S, H \rangle$ and write $\langle i, f, s, h \rangle : \langle F, S, H \rangle$, if the following conditions are satisfied.

- $0 \leq i < |P|$.
- For each variable index y , $\langle f[y], h \rangle : F_i[y]$.
- For each index p for s , $\langle s[p], h \rangle : S_i[p]$.
- $h \in H_i$.
- h does not have duplication, i.e., no element of h occurs more than once in h .

We have the following correctness theorem. It says that if F , S and H follow the rule for each instruction of P , then the soundness is preserved under the transition of quadruples. This means that if the initial quadruple is sound, then quadruples that appear during execution of the virtual machine are always sound.

Theorem (correctness of analysis): Assume that F , S and H follow the rule for each instruction of P . If $\langle i, f, s, h \rangle : \langle F, S, H \rangle$ and $\langle i, f, s, h \rangle \rightarrow \langle i', f', s', h' \rangle$, then $\langle i', f', s', h' \rangle : \langle F, S, H \rangle$.

The theorem is proved by the case analysis on the kind of $P[i]$. In this short paper, we only examine the case when $P[i] = \text{ret}(x)$.

Assume that $\langle i, f, s, h \rangle : \langle F, S, H \rangle$ and $\langle i, f, s, h \rangle \rightarrow \langle i', f', s', h' \rangle$. Since F , S and H follow the rule for **ret**, the following facts hold.

- (i) $F_i[x] = \text{return}(n)$.

(ii) $h = h_1 @ [j] @ h_2$, where $|h_1| = n - 1$.

(iii) $0 \leq j + 1 < |P|$.

(iv) For each variable index y ,

$$F_{j+1}[y] \geq \text{follow_last}(n, h, F_i[y]).$$

(v) $S_{j+1} \geq \text{follow_last}(n, h, S_i)$.

(vi) $h_2 \in H_{j+1}$.

By (i) and the soundness of $\langle i, f, s, h \rangle, \langle f[x], h \rangle : \mathbf{return}(n)$. Therefore, by (ii), $f[x] = \mathbf{retaddr}(j+1)$ and $i' = j+1$. Moreover, since h does not have duplication, h_1 does not contain j . This implies that $h' = h_2$. We also have that $f' = f$ and $s' = s$.

Let us check the conditions for the soundness of $\langle i', f', s', h' \rangle = \langle j+1, f, s, h_2 \rangle$.

- By (iii), $0 \leq i' < |P|$.
- By (iv), $F_{i'}[y] \geq \text{follow_last}(n, h, F_i[y])$. By the soundness of $\langle i, f, s, h \rangle, \langle f[y], h \rangle : F_i[y]$. By Lemma 2, $\langle f[y], h' \rangle : \text{follow_last}(n, h, F_i[y])$. Therefore, by Lemma 1, $\langle f[y], h' \rangle : F_{i'}[y]$.
- Similarly, by (v), we have that $\langle s[p], h' \rangle : S_{i'}[p]$.
- By (vi) and since $h' = h_2$, $h' \in H_{i'}$.
- Finally, since h does not have duplication, h' does not have duplication, either.

Proposition 2: If $\langle i, f, s, h \rangle : \langle F, S, H \rangle$ and $\langle i, f, s \rangle \rightarrow \langle i', f', s' \rangle$, then there exists some h' such that $\langle i, f, s, h \rangle \rightarrow \langle i', f', s', h' \rangle$.

The only case that must be examined is that of **ret**. Note that h' is uniquely determined.

The above proposition guarantees that if F, S and H follow the rule for each instruction and the initial quadruple $\langle i, f, s, h \rangle$ is sound, then the transition sequence starting from the triple $\langle i, f, s \rangle$ can always be lifted to a sequence starting from $\langle i, f, s, h \rangle$. This means that the semantics for triples and that for quadruples coincide when F, S and H follow the rule for each instruction. A similar lemma is stated in [16], which establishes a correspondence between their stackless semantics and their structured semantics.

Lemma 3: If $\langle i, f, s, h \rangle : \langle F, S, H \rangle$, then $\langle i, f, s, h \rangle$ has the next state unless $P[i] = \mathbf{halt}$.

The following final theorem, derived from the above lemma and the previous theorem, guarantees the type safety of bytecode. This corresponds to Theorem 1 (Soundness) in [16].

Theorem (type safety): If F, S and H follow the rule for each instruction and the initial quadruple $\langle i, f, s, h \rangle$ is sound, then a transition sequence stops only at the **halt** instruction.

4.5 Example

Let us abbreviate `last(x)` and `return(n)` by $l(x)$ and $r(n)$, respectively. Following is the result of analyzing the example in Sect. 2.

i instruction	$[F_i[0], F_i[1], F_i[2]]$	S_i	H_i
0 <code>const0</code>	$[\top, \top, \top]$	\square	$\{\square\}$
1 <code>store(1)</code>	$[\top, \top, \top]$	$[\text{INT}]$	$\{\square\}$
2 <code>jsr(7)</code>	$[\top, \text{INT}, \top]$	\square	$\{\square\}$
3 <code>constNULL</code>	$[\top, \text{INT}, \text{INT}]$	\square	$\{\square\}$
4 <code>store(1)</code>	$[\top, \text{INT}, \text{INT}]$	$[\text{OBJ}]$	$\{\square\}$
5 <code>jsr(7)</code>	$[\top, \text{OBJ}, \text{INT}]$	\square	$\{\square\}$
6 <code>halt</code>	$[\top, \text{OBJ}, \text{OBJ}]$	\square	$\{\square\}$
7 <code>store(0)</code>	$[l(0), l(1), l(2)]$	$[r(1)]$	$\{[2], [5]\}$
8 <code>load(1)</code>	$[r(1), l(1), l(2)]$	\square	$\{[2], [5]\}$
9 <code>store(2)</code>	$[r(1), l(1), l(2)]$	$[l(1)]$	$\{[2], [5]\}$
10 <code>ret(0)</code>	$[r(1), l(1), l(1)]$	\square	$\{[2], [5]\}$

The rule for `ret(0)` at 10 is satisfied because, for $[2] \in H_{10}$,

- $F_{10}[0] = \text{return}(1)$. $[2] = \square @ [2] @ \square$. $0 \leq 2+1 = 3 < 11$.
- $F_3[0] = \top \geq \text{follow_last}(1, [2], F_{10}[0]) = \text{follow_last}(1, [2], \text{return}(1)) = \top$.
- $F_3[1] = \text{INT} \geq \text{follow_last}(1, [2], F_{10}[1]) = \text{follow_last}(1, [2], \text{last}(1)) = F_2[1] = \text{INT}$.
- $F_3[2] = \text{INT} \geq \text{follow_last}(1, [2], F_{10}[2]) = \text{follow_last}(1, [2], \text{last}(1)) = F_2[1] = \text{INT}$.
- $S_3 = \square$. $\square \in \{\square\} = H_3$,

and similarly for $[5] \in H_{10}$.

4.6 Returning to an Outer Caller

When $P[i] = \text{ret}(x)$ returns to the caller of the caller, for example, $F_i[x]$ must be equal to the type `return(2)`. In this case, H_i should consist of histories of length at least 2.

If $[j_1, j_2, j_3, \dots]$ is in H_i , $P[i]$ returns to $i' = j_2+1$ and $F_{i'}$ should satisfy

$$F_{i'}[y] \geq \text{follow_last}(2, [j_1, j_2, j_3, \dots], F_i[y]).$$

If $F_i[y] = \text{last}(y)$ and $F_{j_1} = \text{last}(y)$, for example, then

$$\text{follow_last}(2, [j_1, j_2, j_3, \dots], F_i[y]) = F_{j_2}[y].$$

This is how information at j_2 (which is a `jsr`) is propagated to $i' = j_2+1$. If $F_i[y]$ is not `last`, information at i is propagated.

5 Implementation

A dataflow analysis is usually implemented by an iterative algorithm. For each instruction, we check if the rule for the instruction is satisfied by F , S and H . If not, we update F , S and H accordingly, and check the next instruction that is affected by the update.

There are two problems for implementing our analysis by such an iterative algorithm. Firstly, the rule for **jsr** does not uniquely determine $F_L[y]$ when $F_i[y]$ is not **last**. We have two choices: one is to set $F_L[y] = \mathbf{last}(y)$, and the other is to set $F_L[y] = F_i[y]$ (or $F_L[y] = \mathbf{return}(n+1)$ if $F_i[y] = \mathbf{return}(n)$). In our current implementation, we first set $F_L[y] = \mathbf{last}(y)$ and proceed the analysis. If the analysis fails at some point because $F_L[y] = \mathbf{last}(y)$, we take the alternative and redo the analysis from L . (We need not completely abandon the work after we set $F_L[y] = \mathbf{last}(y)$.)

The second problem is that by a naïve iterative algorithm, a subroutine is analyzed each time it is called. In the worst case this may require an exponential number of steps with respect to the length of the program. This problem can be avoided by representing invocation histories by a node in the call graph of the program, which is a graph whose nodes are addresses of subroutines and whose (directed) edges are labeled with addresses of **jsr** instructions. Since the JVM does not allow recursion, it is a connected acyclic graph with a unique root node representing the initial address.

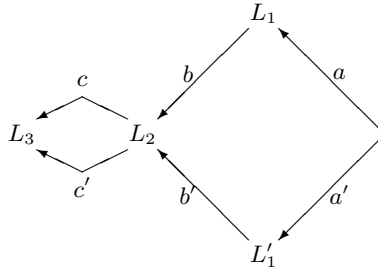
A path from the root to a node in the graph corresponds to an invocation history by concatenating the labels of edges in the path. Each node in the graph then represents the set of all the invocation histories from the root to the node. Now, instead of keeping a set of invocation histories (i.e., H_i), we can keep a set of nodes in the graph.

From a program in the following (left), the call graph in the right is constructed. The node L_3 represents the set $\{[c, b, a], [c, b', a'], [c', b, a], [c', b', a']\}$ of invocation histories.

```

a : jsr(L1)
...
a' : jsr(L'1)
...
L1 : ...
  b : jsr(L2)
  ...
L'1 : ...
  b' : jsr(L2)
  ...
L2 : ...
  c : jsr(L3)
  ...
  c' : jsr(L3)
  ...
L3 : ...

```



If histories are represented by nodes in the call graph, then the values that H_i can take are bounded by the set of all the nodes in the call graph. This means that H_i can only be updated for the number of times equal to the number of nodes. The number of overall updates is, therefore, limited by n^2 , where n is the number of instructions in the program. In order to achieve a polynomial complexity of the entire analysis, however, the nondeterministic choice in the handling of `jsr` instructions must be restricted.

6 Concluding Remark

Since we introduced types of the form `last(x)`, it has become possible to assign types to polymorphic subroutines that move a value from a variable to another. Our analysis is towards the real polymorphism of subroutines in binary code, because we do not simply ignore unaccessed variables. For some programs, our method is more powerful than existing ones. In particular, we impose no restrictions on the number of entries and exits of a subroutine. It is also important that the proof of the correctness of bytecode verification becomes extremely simple.

We only formalized a very small subset of the JVM. We believe that the framework of the paper can be extended to the full language. The extension is almost straight forward. In particular, adding new kinds of types seems to cause no difficulty. The correct handling of exceptions and that of object initialization are problematic but are not impossible [13, 5]. The resulting bytecode verifier is expected to be more powerful than the existing one, so the Java compiler will gain more freedom in bytecode generation.

It is also interesting whether the framework can be applied to the analysis of other kinds of bytecode or native code. Handling recursive calls is the key to such applications. In order to allow recursion, we must be able to represent histories of an indefinite length by a kind of regular expression. Stacks generated by recursive calls should also be represented by regular expressions. All this is left as future work.

A dataflow analysis, in general, assigns an abstract value x_i to the i -th instruction so that a certain predicate $P(x_i, \sigma)$ always holds for any state σ that reaches the i -th instruction. To this end, for any transition $\sigma \rightarrow \sigma'$, where σ' is at i' , one must show that $P(x_i, \sigma)$ implies $P(x_{i'}, \sigma')$.

Since σ corresponds to $\langle i, f, s, h \rangle$ in our analysis, x_i seems to correspond to $\langle F_i, S_i, H_i \rangle$. However, the predicate $\langle i, f, s, h \rangle : \langle F, S, H \rangle$, which should correspond to $P(x_i, \sigma)$, does not only refer to $\langle F_i, S_i, H_i \rangle$. When $F_i[y]$ is `last`, it also refers to $F_{h[0]}$. This means that in terms of `last` types, our analysis relates values assigned to different instructions. This makes the analysis powerful while keeping the overall data structure for the analysis compact.

The representation of invocation histories by a node in the call graph is also for making the data structure small and efficient. By this representation, the number of updates of H_i is limited by the size of the call graph, and an iterative algorithm is expected to stop in polynomial time with respect to the program size. This kind of complexity analysis should be made rigorous in the future.

Acknowledgments. The author would like to thank Zhenyu Qian and Martín Abadi for their comments on the earlier draft of this paper. He also thanks anonymous referees whose comments greatly improved the paper.

References

1. Richard M. Cohen: *The Defensive Java Virtual Machine Specification*, Version Alpha 1 Release, DRAFT VERSION, 1997.
<http://www.cli.com/software/djvm/html-0.5/djvm-report.html>
2. Drew Dean: The Security of Static Typing with Dynamic Linking, *Fourth ACM Conference on Computer and Communication Security*, 1997, pp.18–27.
<http://www.cs.princeton.edu/sip/pub/ccs4.html>
3. Sophia Drossopoulou and Susan Eisenbach: Java is Type Safe — Probably, *ECOOP’97 — Object-Oriented Programming*, Lecture Notes in Computer Science, Vol.1241, 1997, pp.389–418.
<http://outoften.doc.ic.ac.uk/projects/slurp/papers.html#ecoop>
4. Sophia Drossopoulou, Susan Eisenbach and Sarfraz Khurshid: Is the Java Type System Sound? *Proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages*, 1997.
<http://outoften.doc.ic.ac.uk/projects/slurp/papers.html#tapos>
5. Allen Goldberg: A Specification of Java Loading and Bytecode Verification, 1997.
<http://www.kestrel.edu/~goldberg/>
6. James Gosling, Bill Joy and Guy Steele: *The Java™ Language Specification*, Addison-Wesley, 1996.
7. Kimera: <http://kimera.cs.washington.edu/>
8. Tim Lindholm and Frank Yellin: *The Java™ Virtual Machine Specification*, Addison-Wesley, 1997.
9. Gary McGraw and Edward W. Felten: *Java Security: Hostile Applets, Holes and Antidotes*, John Wiley and Sons, 1996.
10. George C. Necula: Proof-Carrying Code, *the Proceedings of the 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, pp.106–117.
11. George C. Necula, Peter Lee: The Design and Implementation of a Certifying Compiler, submitted to *PLDI’98*.
12. Tobias Nipkow and David von Oheimb: Java_{light} is Type-Safe — Definitely, *Proceedings of the 25th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998, pp.161–170.
13. Zhenyu Qian: A Formal Specification of Java™ Virtual Machine Instructions, 1997.
<http://www.informatik.uni-bremen.de/~qian/abs-fsjvm.html>
14. Vijay Saraswat: Java is not type-safe, 1997.
<http://www.research.att.com/~vj/bug.html>
15. Secure Internet Programming: <http://www.cs.princeton.edu/sip/>
16. Raymie Stata and Martín Abadi: A Type System for Java Bytecode Subroutines, *Proceedings of the 25th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998, pp.149–160.
17. Don Syme: Proving Java Type Soundness, 1997.
<http://www.cl.cam.ac.uk/users/drs1004/java.ps>

Enabling Sparse Constant Propagation of Array Elements via Array SSA Form

Vivek Sarkar¹ and Kathleen Knobe²

¹ IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
vivek@watson.ibm.com

² Compaq Cambridge Research Laboratory
One Kendall Square, Building 700, Cambridge, MA 02139, USA
knobe@crl.dec.com

Abstract. We present a new static analysis technique based on *Array SSA form* [6]. Compared to traditional SSA form, the key enhancement in Array SSA form is that it deals with arrays at the element level instead of as monolithic objects. In addition, Array SSA form improves the ϕ function used for merging scalar or array variables in traditional SSA form. The computation of a ϕ function in traditional SSA form depends on the program's control flow in addition to the arguments of the ϕ function. Our improved ϕ function (referred to as a Φ function) includes the relevant control flow information explicitly as arguments through auxiliary variables that are called *@ variables*.

The @ variables and Φ functions were originally introduced as run-time computations in Array SSA form. In this paper, we use the element-level Φ functions in Array SSA form for enhanced static analysis. We use Array SSA form to extend past algorithms for Sparse Constant propagation (SC) and Sparse Conditional Constant propagation (SCC) by enabling constant propagation through array elements. In addition, our formulation of array constant propagation as a set of data flow equations enables integration with other analysis algorithms that are based on data flow equations.

Keywords: static single assignment (SSA) form, constant propagation, conditional constant propagation, Array SSA form, unreachable code elimination.

1 Introduction

The problems of *constant propagation* and *conditional constant propagation* (a combination of constant propagation and unreachable code elimination) have been studied for several years. However, past algorithms limited their attention to constant propagation of scalar variables only. In this paper, we introduce efficient new algorithms that perform constant propagation and conditional constant propagation through both scalar and array references.

One motivation for constant propagation of array variables is in optimization of scientific programs in which certain array elements can be identified as constant. For example, the SPEC95fp [3] benchmark 107.mgrid contains an array variable *A* that is initialized to four constant-valued elements as shown in Fig. 1. A significant amount of the time in this application is spent in the triply nested loop shown at the bottom of Fig. 1. Since constant propagation can determine that *A*(2) equals zero in the loop, an effective optimization is to eliminate the entire multiplicand of *A*(2) in the loop nest. Doing so eliminates 11 of the 23 floating-point additions in the loop nest thus leading to a significant speedup.

Another motivation is in analysis and optimization of field accesses of structure variables or objects in object-oriented languages such as Java and C++. A structure can be viewed as a fixed-size array, and a read/write operation of a structure field can be viewed as a read/write operation of an array element through a subscript that is a compile-time constant. This approach is more compact than an approach in which each field of a structure is modeled as a separate scalar variable. This technique for modeling structures as arrays directly extends to nested arrays and structures. For example, an array of rank n of some structure type can be modeled as an array of rank $n + 1$. Therefore, the constant propagation algorithms presented in this paper can be efficiently applied to structure variables and to arrays of structures. Extending these algorithms to analyze programs containing pointer aliasing is a subject for future research, however.

The best known algorithms for sparse constant propagation of scalar variables [8,2] are based on static single assignment (SSA) form [4]. However, traditional SSA form views arrays as monolithic objects, which is an inadequate view for analyzing and optimizing programs that contain reads and writes of individual array elements. In past work, we introduced Array SSA form [6] to address this deficiency. The primary application of Array SSA form in [6] was to enable parallelization of loops not previously parallelizable by making Array SSA form manifest at run-time. In this paper, we use Array SSA form as a basis for static analysis, which means that the Array SSA form structures can be removed after the program properties of interest have been discovered.

Array SSA form has two distinct advantages over traditional SSA form. First, the ϕ operator in traditional SSA form is not a pure function and returns different values for the same arguments depending on the control flow path that was taken. In contrast, the corresponding Φ operator in Array SSA form includes *@ variables* as extra arguments to capture the control information required *i.e.*, $x_3 := \phi(x_2, x_1)$ in traditional SSA form becomes $x_3 := \Phi(x_2, @x_2, x_1, @x_1)$ in

```

! Initialization of array A
REAL*8 A(0:3)
. . .
A(0) = -8.0D0/3.0D0
A(1) = 0.0D0
A(2) = 1.0D0/6.0D0
A(3) = 1.0D0/12.0D0

. . .

! Computation loop in subroutine RESID()

do i3 = 2, n-1
  do i2 = 2, n-1
    do i1 = 2, n-1
      R(i1,i2,i3)=V(i1,i2,i3)
      -A(0)*( U(i1, i2, i3 ) )
      -A(1)*( U(i1-1,i2, i3 ) + U(i1+1,i2, i3 )
               + U(i1, i2-1,i3 ) + U(i1, i2+1,i3 )
               + U(i1, i2, i3-1) + U(i1, i2, i3+1) )
      -A(2)*( U(i1-1,i2-1,i3 ) + U(i1+1,i2-1,i3 )
               + U(i1-1,i2+1,i3 ) + U(i1+1,i2+1,i3 )
               + U(i1, i2-1,i3-1) + U(i1, i2+1,i3-1)
               + U(i1, i2-1,i3+1) + U(i1, i2+1,i3+1)
               + U(i1-1,i2, i3-1) + U(i1-1,i2, i3+1)
               + U(i1+1,i2, i3-1) + U(i1+1,i2, i3+1) )
      -A(3)*( U(i1-1,i2-1,i3-1) + U(i1+1,i2-1,i3-1)
               + U(i1-1,i2+1,i3-1) + U(i1+1,i2+1,i3-1)
               + U(i1-1,i2-1,i3+1) + U(i1+1,i2-1,i3+1)
               + U(i1-1,i2+1,i3+1) + U(i1+1,i2+1,i3+1) )
    end do
  end do
end do

```

Fig. 1. Code fragments from the SPEC95fp 107.mgrid benchmark

Array SSA form. Second, Array SSA form operates on arrays at the element level rather than as monolithic objects. In particular, a Φ operator in Array SSA form, $A_3 := \Phi(A_2, @A_2, A_1, @A_1)$, represents an *element-level merge* of A_2 and A_1 .

Both advantages of Array SSA form are significant for static analysis. The fact that Array SSA form operates at the element-level facilitates transfer of statically derived information across references to array elements. The fact that the Φ is a known pure function facilitates optimization and simplification of the Φ operations.

For convenience, we assume that all array operations in the input program are expressed as reads and writes of individual array elements. The extension to more complex array operations (*e.g.*, as in Fortran 90 array language) is

straightforward, and omitted for the sake of brevity. Also, for simplicity, we will restrict constant propagation of array variables to cases in which both the subscript and the value of an array definition are constant *e.g.*, our algorithm might recognize that a definition $A[k] := i$ is really $A[2] := 99$ and propagate this constant into a use of $A[2]$. The algorithms presented in this paper will not consider a definition to be constant if its subscript has a non-constant value *e.g.*, $A[m] := 99$ where m is not a constant. Performing constant propagation for such references (*e.g.*, propagating 99 into a use of $A[m]$ when legal to do so) is a subject of future work.

The rest of the paper is organized as follows. Section 2 reviews the Array SSA form introduced in [6]. Section 3 presents our extension to the Sparse Constant propagation (SC) algorithm from [8] that enables constant propagation through array elements. It describes how lattice values can be computed for array variables and Φ functions. Section 4 presents our extension to the Sparse Conditional Constant propagation (SCC) algorithm from [8] that enables constant propagation through array elements in conjunction with unreachable code elimination. Section 5 discusses related work, and Sect. 6 contains our conclusions.

2 Array SSA Form

In this section, we describe the Array SSA form introduced in [6]. The goal of Array SSA form is to provide the same benefits for arrays that traditional SSA provides for scalars but, as we will see, it has advantages over traditional SSA form for scalars as well. We first describe its use for scalar variables and then its use for array variables.

The salient properties of traditional SSA form are as follows:

1. Each definition is assigned a unique name.
2. At certain points in the program, new names are generated which combine the results from several definitions. This combining is performed by a ϕ function which determines which of several values to use, based on the flow path traversed.
3. Each use refers to exactly one name generated from either of the two rules above.

For example, traditional SSA form converts the code in Fig. 2 to that in Fig. 3. The $S_3 := \phi(S_1, S_2)$ statement defines S_3 as a new name that represents the merge of definitions S_1 and S_2 . It is important to note that the ϕ function in traditional SSA form is not a pure function of S_1 and S_2 because its value depends on the path taken through the *if* statement. Notice that this path is unknown until runtime and may vary with each dynamic execution of this code.

In contrast to traditional SSA form, the semantics of a Φ function is defined to be a pure function in our Array SSA form. This is accomplished by introducing *@ variables* (pronounced “at variables”), and by rewriting a ϕ function in traditional SSA form such as $\phi(S_1, S_2)$ as a new kind of Φ function,

$\Phi(S_1, @S_1, S_2, @S_2)$. For each static definition S_k , its $@$ variable $@S_k$ identifies the most recent “time” at which S_k was modified by this definition.

For an *acyclic* control flow graph, a static definition S_k may execute either zero times or one time. These two cases can be simply encoded as $@S_k = \text{FALSE}$ and $@S_k = \text{TRUE}$ to indicate whether or not definition S_k was executed. For a control flow graph with *cycles* (loops), a static definition S_k may execute an arbitrary number of times. In general, we need more detailed information for the $@S_k = \text{TRUE}$ case so as to distinguish among different dynamic execution instances of static definition S_k . Therefore, $@S_k$ is set to contain the dynamic *iteration vector* at which the static definition S_k was last executed.

The *iteration vector* of a static definition S_k identifies a single iteration in the iteration space of the set of loops that enclose the definition. Let n be the number of loops that enclose a given definition. For convenience, we treat the outermost region of acyclic control flow in a procedure as a dummy outermost loop with a single iteration. Therefore $n \geq 1$ for each definition. A single point in the iteration space is specified by the iteration vector $\mathbf{i} = (i_1, \dots, i_n)$, which is an n -tuple of iteration numbers one for each enclosing loop. We do not require that the surrounding loops be structured counted loops (*i.e.*, like Fortran DO loops) or that the surrounding loops be tightly nested. Our only assumption is that all loops are single-entry, or equivalently, that the control flow graph is *reducible* [5, 11]. For single-entry loops, we know that each def executes at most once in a given iteration of its surrounding loops. All structured loops (e.g., **do**, **while**, **repeat-until**) are single-entry even when they contain multiple exits; also, most unstructured loops (built out of **goto** statements) found in real programs are single-entry as well. A multiple-entry loop can be transformed into multiple single-entry loops by *node splitting* [5, 11].

Array SSA form can be used either at run-time as discussed in [6] or for static analysis, as in the constant propagation algorithms presented in this paper. In this section, we explain the meaning of $@$ variables as if they are computed at run-time. We assume that all $@$ variables, $@S_k$, are initialized to the empty vector, $@S_k := ()$, at the start of program execution. For each real (non- Φ) definition, S_k , we assume that a statement of the form $@S_k := \mathbf{i}$ is inserted immediately after definition S_k [1], where \mathbf{i} is the current iteration vector for all loops that surround S_k . Each Φ definition also has an associated $@$ variable. Its semantics will be defined shortly. All $@$ variables are initialized to the empty vector because the empty vector is the identity element for a lexicographic max operation *i.e.*, $\max((), \mathbf{i}) = \mathbf{i}$, for any $@$ variable value \mathbf{i} .

As a simple example, Fig. 4 shows the Array SSA form for the program in Fig. 2. Note that $@$ variables $@S_1$ and $@S_2$ are explicit arguments of the Φ function. In this example of acyclic code, there are only two possible values

¹ It may appear that the $@$ variables do not satisfy the static single assignment property because each $@S_k$ variable has two static definitions, one in the initialization and one at the real definition of S_k . However, the initialization def is executed only once at the start of program execution and can be treated as a special-case initial value rather than as a separate definition.

for each @ variable — the empty vector () and the unit vector (1) — which correspond to FALSE and TRUE respectively.

Figure 5 shows an example for-loop and its conversion to Array SSA form. Because of the presence of a loop, the set of possible values for an @ variable becomes unbounded *e.g.*, we may have $@S_1 = (100)$ on exit from the loop. However, $@S_1$ and $@S_2$ are still explicit arguments of the Φ function, and their iteration vector values are necessary for evaluating the Φ function at run-time.

The semantics of a Φ function can now be specified by a conditional expression that is a pure function of the arguments of the Φ . For example, the semantics of the Φ function, $S_3 := \Phi(S_2, @S_2, S_1, @S_1)$ in Fig. 5, can be expressed as a conditional expression as follows (where \succeq denotes a lexicographic greater-than-or-equal comparison of iteration vectors):

$$\begin{array}{ll} \text{if} & @S_2 \succeq @S_1 \text{ then } S_2 \\ S_3 = & \text{else } S_1 \\ \text{end if} & \end{array}$$

Following each Φ def, $S_3 := \Phi(S_2, @S_2, S_1, @S_1)$, there is the definition of the associated @ variable, $@S_3 = \max(@S_2, @S_1)$, where \max represents a *lexicographic maximum* operation of iteration vector values $@S_2, @S_1$.

Consider, for example, a condition C in Fig. 5 that checks if the value of i is even. In this case, definition S_1 is executed in every iteration and definition S_2 is executed only in iterations 2, 4, 6, \dots . For this “even-value” branch condition, the final values of $@S_1$ and $@S_2$ are both equal to (100) if $m = 100$. Since these values satisfy the condition $@S_2 \succeq @S_1$, the conditional expression will yield $S_3 = S_2$.

Consider another execution of the for-loop in Fig. 5 in which condition C evaluates to FALSE in each iteration of the for loop. For this execution, the final values of $@S_2$ and $@S_1$ will be the empty vector () and (100) respectively. Therefore, $S_2 \prec S_1$, and the conditional expression for the Φ function will yield $S_3 = S_1$ for this execution.

The above description outlines how @ variables and Φ functions can be computed at run-time. However, if Array SSA form is used for static analysis, then no run-time overhead is incurred due to the @ variables and Φ functions. Instead, the @ variables and Φ functions are inserted in the compiler intermediate representation prior to analysis, and then removed after the program properties of interest have been discovered by static analysis.

We now describe Array SSA form for array variables. Figure 6 shows an example program with an array variable, and the conversion of the program to Array SSA form as defined in [6]. The key differences between Array SSA form for array variables and Array SSA form for scalar variables are as follows:

1. Array-valued @ variables:

The @ variable is an array of the same shape as the array variable with which it is associated, and each element of an @ array is initialized to the empty vector. For example, the statement $@A_1[k_1] := (1)$ is inserted after


```

if (C) then
  S := ...
else
  S := ...
end if

```

Fig. 2. Control Flow with Scalar Definitions

```

if (C) then
  S1 := ...
else
  S2 := ...
end if
S3 :=  $\phi(S_1, S_2)$ 

```

Fig. 3. Traditional SSA form

```

@S1 := ( )
@S2 := ( )

if (C) then
  S1 := ...
  @S1 := (1)
else
  S2 := ...
  @S2 := (1)
end if
S3 =  $\Phi(S_1, @S_1, S_2, @S_2)$ 

```

Fig. 4. After conversion of Fig. 2 to Array SSA form

Example for-loop:

```

S := ...
for i := 1 to m do
  S := ...
  if (C) then
    S := ...
  end if
end for

```

After conversion to Array SSA form:

```

@S1 := ( ) ; @S2 := ( )
S := ...
@S := (1)
for i := 1 to m do
  S0 :=  $\Phi(S_3, @S_3, S, @S)$ 
  @S0 := max(@S3, @S)
  S1 := ...
  @S1 := (i)
  if (C) then
    S2 := ...
    @S2 := (i)
  end if
  S3 :=  $\Phi(S_2, @S_2, S_1, @S_1)$ 
  @S3 := max(@S2, @S2)
end for

```

Fig. 5. A for-loop and its conversion to Array SSA form

Example program with array variables:

```

n1:  A[*] := initial value of A
      i := 1
      C := i < n
      if C then
n2:    k := 2 * i
        A[k] := i
        print A[k]
      endif
n3:  print A[2]

```

After conversion to Array SSA form:

```

n1:  @i := ( ) ; @C := ( ) ; @k := ( ) ; @A0[*] := ( ) ; @A1[*] := ( )

      A0[*] := initial value of A
      @A0[*] := (1)
      i := 1
      @i := (1)
      C := i < n
      @C := (1)
      if C then
n2:    k := 2 * i
        @k := (1)
        A1[k] := i
        @A1[k] := (1)
        A2 := dΦ(A1, @A1, A0, @A0)
        @A2 := max(@A1, @A0)
        print A2[k]
      endif
n4:  A3 := Φ(A2, @A2, A0, @A0)
      @A3 := max(@A2, @A0)
      print A3[2]

```

Fig. 6. Example program with an array variable, and its conversion to Array SSA form

statement $A_1[k_1] := i$ in Fig. 6. In general, $@A_1$ can record a separate iteration vector for each element that is assigned by definition A_1 . This initialization is only required for $@$ arrays corresponding to real (non- Φ) definitions. No initialization is required for an $@$ array for a Φ definition (such as $@A_2$ and $@A_3$ in Fig. 6) because its value is completely determined by other $@$ arrays.

2. Array-valued Φ functions:

A Φ function for array variables returns an array value. For example, consider the Φ definition $A_3 := \Phi(A_2, @A_2, A_0, @A_0)$ in Fig. 6 which represents a merge of arrays A_2 and A_0 . The semantics of the Φ function is specified by the following conditional expression for each element, $A_3[j]$:

$$A_3[j] = \begin{array}{ll} \text{if} & @A_2[j] \succeq @A_0[j] \text{ then } A_2[j] \\ \text{else} & A_0[j] \\ \text{end if} \end{array}$$

Note that this conditional expression uses a lexicographic comparison (\succeq) of $@$ values just as in the scalar case.

3. Definition Φ 's:

The traditional placement of the Φ is at control merge points. We refer to this as a *control* Φ . A special new kind of Φ function is inserted immediately after each original program definition of an array variable that does not completely kill the array value. This *definition* Φ merges the value of the element modified in the definition with the values available immediately prior to the definition. Definition Φ 's did not need to be inserted for definitions of scalar variables because a scalar definition completely kills the old value of the variable. We will use the notation $d\Phi$ when we want to distinguish a definition Φ function from a control Φ function.

For example, consider definition A_1 in Fig. 6. The $d\Phi$ function, $A_2 := d\Phi(A_1, @A_1, A_0, @A_0)$ is inserted immediately after the def of A_1 to represent an element-by-element merge of A_1 and A_0 . Any subsequent use of the original program variable A (before an intervening def) will now refer to A_2 instead of A_1 . The semantics of the $d\Phi$ function is specified by the following conditional expression for each element, $A_2[j]$:

$$A_2[j] = \begin{array}{ll} \text{if} & @A_1[j] \succeq @A_0[j] \text{ then } A_1[j] \\ \text{else} & A_0[j] \\ \text{end if} \end{array}$$

Note that this conditional expression is identical in structure to the conditional expression for the control Φ function in item 2 above.

3 Sparse Constant Propagation for Scalars and Array Elements

We now present our extension to the Sparse Constant propagation (SC) algorithm from 8 that enables constant propagation through array elements.

Section 3.1 contains our definitions of lattice elements for scalar and array variables; the main extension in this section is our modeling of lattice elements for array variables. Section 3.2 outlines our sparse constant propagation algorithm; the main extension in this section is the use of the definition Φ operator in Array SSA form to perform constant propagation through array elements.

3.1 Lattice Values for Scalar and Array Variables

Recall that a lattice consists of:

- \mathcal{L} , a set of lattice elements. A lattice element for a program variable v is written as $\mathcal{L}(v)$, and denotes $\text{SET}(\mathcal{L}(v)) = \text{a set of possible values for variable } v$.
- \top (“top”) and \perp (“bottom”), two distinguished elements of \mathcal{L} .
- A *meet* (or *join*) operator, \sqcap , such that for any lattice element e , $e \sqcap \top = e$ and $e \sqcap \perp = \perp$.
- A \sqsupseteq operator such that $e \sqsupseteq f$ if and only if $e \sqcap f = f$, and a \sqsubset operator such that $e \sqsubset f$ if and only if $e \sqsupseteq f$ and $e \neq f$.

The *height* H of lattice \mathcal{L} is the length of the largest sequence of lattice elements e_1, e_2, \dots, e_H such that $e_i \sqsubset e_{i+1}$ for all $1 \leq i < H$.

We use the same approach as in [8] for modeling lattice elements for scalar variables. Given a scalar variable S , the value of $\mathcal{L}(S)$ in our framework can be \top , *Constant* or \perp . When the value is *Constant* we also maintain the value of the constant. The sets denoted by these lattice elements are $\text{SET}(\top) = \{ \}$, $\text{SET}(\text{Constant}) = \{\text{Constant}\}$, and $\text{SET}(\perp) = \mathcal{U}^S$, where \mathcal{U}^S is the universal set of values for variable S .

We now describe how lattice elements for array variables are represented in our framework. Let \mathcal{U}_{ind}^A and \mathcal{U}_{elem}^A be the universal set of *index values* and the universal set of array *element values* respectively for an array variable A in Array SSA form. For an array variable, the set denoted by lattice element $\mathcal{L}(A)$ is a subset of $\mathcal{U}_{ind}^A \times \mathcal{U}_{elem}^A$ i.e., a set of index-element pairs. Since we restrict constant propagation of array variables to cases in which both the subscript and the value of an array definition are constant, there are only three kinds of lattice elements of interest:

1. $\mathcal{L}(A) = \top \Rightarrow \text{SET}(\mathcal{L}(A)) = \{ \}$

This “top” case means that the possible values of A have yet to be determined i.e., the set of possible index-element pairs that have been identified thus far for A is the empty set, $\{ \}$.

2. $\mathcal{L}(A) = \langle (i_1, e_1), (i_2, e_2), \dots \rangle$

$$\Rightarrow \text{SET}(\mathcal{L}(A)) = \{ (i_1, e_1), (i_2, e_2), \dots \} \cup (\mathcal{U}_{ind}^A - \{i_1, i_2, \dots\}) \times \mathcal{U}_{elem}^A$$

In general, the lattice value for this “constant” case is represented by a finite ordered list of index-element pairs, $\langle (i_1, e_1), (i_2, e_2), \dots \rangle$ where $i_1, e_1, i_2, e_2, \dots$ are all constant. The list is sorted in ascending order of the index values, i_1, i_2, \dots , and all the index values assumed to be distinct.

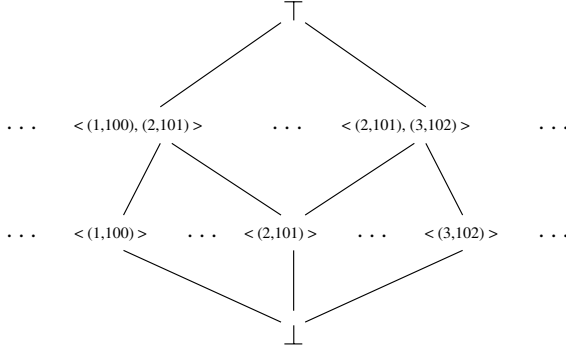


Fig. 7. Lattice elements of array values with maximum list size $Z = 2$

The meaning of this “constant” lattice value is that the current stage of analysis has determined some finite number of constant index-element pairs for array variable A , such that $A[i_1] = e_1$, $A[i_2] = e_2$, \dots . All other elements of A are assumed to be non-constant. These properties are captured by $\text{SET}(\mathcal{L}(A))$ defined above as the set denoted by lattice value $\mathcal{L}(A)$.

For the sake of efficiency, we will restrict these constant lattice values to ordered lists that are bounded in size by a small constant, $Z \geq 1$ *e.g.*, if $Z = 5$ then all constant lattice values will have ≤ 5 index-element pairs. Doing so ensures that the height of the lattice for array values is at most $(Z + 2)$. If any data flow equation yields a lattice value with $P > Z$ pairs, then this size constraint is obeyed by conservatively dropping any $(P - Z)$ index-element pairs from the ordered list.

Note that the lattice value for a real (non- Φ) definition, will contain at most one index-element pair, since we assumed that an array assignment only modifies a single element. Ordered lists with size > 1 can only appear as the output of Φ functions.

3. $\mathcal{L}(A) = \perp \Rightarrow \text{SET}(\mathcal{L}(A)) = \mathcal{U}_{ind}^A \times \mathcal{U}_{elem}^A$

This “bottom” case means that, according to the approximation in the current stage of analysis, array A may take on any value from the universal set of index-element pairs. Note that $\mathcal{L}(A) = \perp$ is equivalent to an empty ordered list, $\mathcal{L}(A) = \langle \rangle$.

The lattice ordering (\sqsubset) for these elements is determined by the subset relationship among the sets that they denote. The lattice structure for the $Z = 2$ case is shown in Fig. 7. This lattice has four levels. The second level (just below \top) contains all possible ordered lists that contain exactly two constant index-element pairs. The third level (just above \perp) contains all possible ordered lists that contain a single constant index-element pair. The lattice ordering is determined by the subset relationship among the sets denoted by lattice elements. For example, consider two lattice elements $\mathcal{L}_1 = \langle (1, 100), (2, 101) \rangle$

and $\mathcal{L}_2 = \langle (2, 101) \rangle$. The sets denoted by these lattice elements are:

$$\begin{aligned}\text{SET}(\mathcal{L}_1) &= \{(1, 100), (2, 101)\} \cup (\mathcal{U}_{ind} - \{1, 2\}) \times \mathcal{U}_{elem} \\ \text{SET}(\mathcal{L}_2) &= \{(2, 101)\} \cup (\mathcal{U}_{ind} - \{2\}) \times \mathcal{U}_{elem}\end{aligned}$$

Therefore, $\text{SET}(\mathcal{L}_1)$ is a proper subset of $\text{SET}(\mathcal{L}_2)$ and we have $\mathcal{L}_1 \sqsubset \mathcal{L}_2$ *i.e.*, \mathcal{L}_1 is above \mathcal{L}_2 in the lattice in Fig. 7.

Finally, the meet operator (\sqcap) for two lattice elements, \mathcal{L}_1 and \mathcal{L}_2 , for array variables is defined in Fig. 8 where $\mathcal{L}_1 \cap \mathcal{L}_2$ denotes an intersection of ordered lists \mathcal{L}_1 and \mathcal{L}_2 .

$\mathcal{L}_3 = \mathcal{L}_1 \sqcap \mathcal{L}_2$	$\mathcal{L}_2 = \top$	$\mathcal{L}_2 = \langle (i_1, e_1), \dots \rangle$	$\mathcal{L}_2 = \perp$
$\mathcal{L}_1 = \top$	\top	\mathcal{L}_2	\perp
$\mathcal{L}_1 = \langle (i'_1, e'_1), \dots \rangle$	\mathcal{L}_1	$\mathcal{L}_1 \cap \mathcal{L}_2$	\perp
$\mathcal{L}_1 = \perp$	\perp	\perp	\perp

Fig. 8. Lattice computation for the meet operator, $\mathcal{L}_3 = \mathcal{L}_1 \sqcap \mathcal{L}_2$

3.2 The Algorithm

Recall that the @ variables defined in Sect. 2 were necessary for defining the full execution semantics of Array SSA form. For example, the semantics of a Φ operator, $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$, is defined by the following conditional expression:

$$\begin{aligned} & \text{if } @A_1[j] \succeq @A_0[j] \text{ then } A_1[j] \\ A_2[j] &= \text{else } A_0[j] \\ & \text{end if} \end{aligned}$$

The sparse constant propagation algorithm presented in this section is a static analysis that is based on conservative assumptions about runtime behavior. Let us first consider the case when the above Φ operator is a control Φ . Since algorithm in this section does not perform conditional constant propagation, the lattice computation of a control Φ can be simply defined as $\mathcal{L}(A_2) = \mathcal{L}(\Phi(A_1, @A_1, A_0, @A_0)) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$ *i.e.*, as a join of the lattice values $\mathcal{L}(A_1)$ and $\mathcal{L}(A_0)$. Therefore, the lattice computation for A_2 does not depend on @ variables $@A_1$ and $@A_0$ for a control Φ operator.

Now, consider the case when the above Φ operator is a definition Φ . The lattice computation for a definition Φ is shown in Fig. 9. Since A_1 corresponds to a definition of a single array element, the ordered list for $\mathcal{L}(A_1)$ can contain at most one pair. The INSERT operation in Fig. 9 is assumed to return a new ordered list obtained by inserting (i', e') into $\langle (i_1, e_1), \dots \rangle$ with the following adjustments if needed:

- If there exists an index-element pair (i_j, e_j) in $\langle (i_1, e_1), \dots \rangle$ such that $i' = i_j$, then the INSERT operation just replaces (i_j, e_j) by (i', e') .
- If the INSERT operation causes the size of the list to exceed the threshold size Z , then one of the pairs is dropped from the output list so as to satisfy the size constraint.

Interestingly, we again do not need @ variables for the lattice computation in Fig. 9. This is because the ordered list representation for array lattice values already contains all the subscript information of interest, and overlaps with the information that would have been provided by @ variables.

$\mathcal{L}(A_2)$	$\mathcal{L}(A_0) = \top$	$\mathcal{L}(A_0) = \langle (i_1, e_1), \dots \rangle$	$\mathcal{L}(A_0) = \perp$
$\mathcal{L}(A_1) = \top$	\top	\top	\top
$\mathcal{L}(A_1) = \langle (i', e') \rangle$	\top	$\text{INSERT}((i', e'), \langle (i_1, e_1), \dots \rangle)$	$\langle (i', e') \rangle$
$\mathcal{L}(A_1) = \perp$	\perp	\perp	\perp

Fig. 9. Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{d\Phi}(\mathcal{L}(A_1), \mathcal{L}(A_0))$

Therefore, we do not need to analyze @ variables for the sparse constant propagation algorithm described in this section, because of our ordered list representation of lattice values for array variables. Instead, we can use a *partial* Array SSA form which is simply Array SSA form with all definitions and uses of @ variables removed, and with ϕ operators instead of Φ operators. If only constant propagation is being performed, then it would be more efficient to only build the partial Array SSA form. However, if other optimizations are being performed that use Array SSA form, then we can build full Array SSA form and simply ignore the @ variables for this particular analysis.

Our running example is shown in Fig. 10. The partial Array SSA form for this example is shown in Fig. 11. The partial Array SSA form does not contain any @ variables since @ variables are not necessary for the level of analysis performed by the constant propagation algorithms in this paper. The data flow equations for this example are shown in Fig. 12. Each assignment in the Array SSA form results in one data flow equation. The numbering S1 through S8 indicates the correspondence.

The argument to these equations are simply the current lattice values of the variables. The lattice operations are specific to the operations within the statement. Figures 13, 9, and 14 show the lattice computations for an assignment to an array element (as in S3 and S5), definition ϕ (as in S4 and S6), a reference to an array element (as in the RHS of S3 and S5). The lattice computation for a ϕ assignment (as in S7) $A_3 = \phi(A_2, A_1)$ is $\mathcal{L}(A_3) = \mathcal{L}_\phi(\mathcal{L}(A_2), \mathcal{L}(A_1)) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_2)$ where \sqcap is shown in Fig. 8. Notice that we also include lattice computation for specific arithmetic computations such as the multiply in S3 and S5. This allows for constant computation as well as constant propagation. Tables for these arithmetic computations are straightforward and are not shown.

The simple data flow algorithm is shown in Fig. 15. Our example includes the propagation of values from a node to two successors (Y) and to a node from two predecessors (D). Equations S1 and S2 are evaluated because they are associated with the entry block. Element 3 of Y_2 is known to have the value 99 at this stage. As a result of the modification of Y_2 both S3 and S5 are inserted into the worklist (they reference the lattice value of Y_2). S3 uses the propagated constant 99 to compute and propagate the constant 198 to element 1 of D_1 and then, after evaluation of S4, to element 1 of D_2 . Any subsequent references to D in the **then** block of the source become references D_2 in the Array SSA form and are known to have a constant value at element 1. Depending on the order of computations via the worklist, we may then compute either D_3 and then D_4 or, because D_2 has been modified, we may compute D_5 . If we compute D_5 at this point, it appears to have a constant value. Subsequent evaluations D_3 and D_4 cause D_5 to be reevaluated and lowered from constant value to \perp because the value along one path is not constant.

Notice that in this case, the reevaluation of D_5 could have been avoided by choosing an optimal ordering of processing. Processing of programs with cyclic control flow is no more complex but may involve recomputation that can not be removed by optimal reordering. In particular, the loop entry is a control flow merge point since control may enter from the top or come from the loop body. It will contain a ϕ which combines the value entering from the top with that returning after the loop. The lattice values for such a node may require multiple evaluations.

Also notice that in this example, if I in S5 is known to have the value 3, it will be recoded as a constant element. Upon evaluation of S7, the intersection of the sets associated with D_2 and D_4 will not be empty and element 1 of D_5 will be recorded as a constant.

```

Y[3] := 99
if C then
    D[1] := Y[3] * 2
else
    D[1] := Y[I] * 2
endif
Z := D[1]

```

Fig. 10. Sparse Constant Propagation Example

Y_0 and D_0 in effect here.

```

...
S1:   $Y_1[3] := 99$ 
S2:   $Y_2 := \phi(Y_1, Y_0)$ 
      if  $C$  then
S3:     $D_1[1] := Y_2[3] * 2$ 
S4:     $D_2 := \phi(D_1, D_0)$ 
      else
S5:     $D_3[1] := Y_2[I] * 2$ 
S6:     $D_4 := \phi(D_3, D_0)$ 
      endif
S7:   $D_5 := \phi(D_2, D_4)$ 
S8:   $Z := D_5[1]$ 

```

Fig. 11. Array SSA form for the Sparse Constant Propagation Example

```

S1:  $\mathcal{L}(Y_1) = \langle (3, 99) \rangle$ 
S2:  $\mathcal{L}(Y_2) = \mathcal{L}_{d\phi}(\mathcal{L}(Y_1), \mathcal{L}(Y_0))$ 
S3:  $\mathcal{L}(D_1) = \mathcal{L}_{d[\ ]}(\mathcal{L}_*(\mathcal{L}(Y_2[3]), 2))$ 
S4:  $\mathcal{L}(D_2) = \mathcal{L}_{d\phi}(\mathcal{L}(D_1), \mathcal{L}(D_0))$ 
S5:  $\mathcal{L}(D_3) = \mathcal{L}_{d[\ ]}(\mathcal{L}_*(\mathcal{L}(Y_2[I]), 2))$ 
S6:  $\mathcal{L}(D_4) = \mathcal{L}_{d\phi}(\mathcal{L}(D_3), \mathcal{L}(D_0))$ 
S7:  $\mathcal{L}(D_5) = \mathcal{L}_\phi(\mathcal{L}(D_2), \mathcal{L}(D_4))$ 
S8:  $\mathcal{L}(Z) = \mathcal{L}(D_5[1])$ 

```

Fig. 12. Data Flow Equations for the Sparse Constant Propagation Example

$\mathcal{L}(A_1)$	$\mathcal{L}(i) = \top$	$\mathcal{L}(i) = \text{Constant}$	$\mathcal{L}(i) = \perp$
$\mathcal{L}(k) = \top$	\top	\top	\perp
$\mathcal{L}(k) = \text{Constant}$	\top	$\langle (\mathcal{L}(k), \mathcal{L}(i)) \rangle$	\perp
$\mathcal{L}(k) = \perp$	\perp	\perp	\perp

Fig. 13. Lattice computation for array definition operator, $\mathcal{L}(A_1) = \mathcal{L}_{d[\]}(\mathcal{L}(k), \mathcal{L}(i))$

$\mathcal{L}(A[k])$	$\mathcal{L}(k) = \top$	$\mathcal{L}(k) = \text{Constant}$	$\mathcal{L}(k) = \perp$
$\mathcal{L}(A) = \top$	\top	\top	\perp
$\mathcal{L}(A) = \langle (i_1, e_1), \dots \rangle$	\top	e_j , if $\exists (i_j, e_j) \in \mathcal{L}(A)$ with $i_j = \mathcal{L}(k)$ \perp , otherwise	\perp
$\mathcal{L}(A) = \perp$	\perp	\perp	\perp

Fig. 14. Lattice computation for array reference operator, $\mathcal{L}(A[k]) = \mathcal{L}_{[\]}(\mathcal{L}(A), \mathcal{L}(k))$

Initialization:

$\mathcal{L}(v) \leftarrow \top$ for all local variables, v .
 $insert(E_v, work_list)$ for each equation E_v defining v
 such that v is assigned to in the entry block.

Body:

while ($work_list \neq empty$)
 $E_v \leftarrow remove(work_list)$
 $reevaluate(E_v)$
 $insert(E'_{v'}, work_list)$ for each equation $E'_{v'}$ that uses E_v
end while

Fig. 15. Algorithm for Sparse Constant propagation (SC) for array and scalar variables

4 Sparse Conditional Constant Propagation

4.1 Lattice Values of Executable Flags for Nodes and Edges

As in the SCC algorithm in [8], our array conditional constant propagation algorithm maintains executable flags associated with each node and each edge in the CFG. Flag X_{ni} indicates whether node n_i may be executed, and X_{ei} indicates whether edge e_i may be traversed. The lattice value of an execution flag is either NO or MAYBE, corresponding to unreachable code and reachable code respectively. The lattice value for an execution flag is initialized to NO, and can be lowered to MAYBE in the course of the constant propagation algorithm. In practice, control dependence identities can be used to reduce the number of executable flag variables in the data flow equations *e.g.*, a single flag can be used for all CFG nodes that are control equivalent. For the sake of simplicity, we ignore such optimizations in this paper.

The executable flag of a node is computed from the executable flags of its incoming edges. The executable flag of an edge is computed from the executable flag of its source node and knowledge of the branch condition variable used to determine the execution path from that node. These executable flag mappings are summarized in Fig. 16 for a node n with two incoming edges, $e1$ and $e2$, and two outgoing edges, $e3$ and $e4$. The first function table in Fig. 16 defines the *join operator* \sqcap on executable flags such that $X_n = X_{e1} \sqcap X_{e2}$. We introduce a *true operator* $\mathcal{L}_{\mathcal{T}}$ and a *false operator* $\mathcal{L}_{\mathcal{F}}$ on lattice values such that $X_{e3} = \mathcal{L}_{\mathcal{T}}(X_n, \mathcal{L}(C))$ and $X_{e4} = \mathcal{L}_{\mathcal{F}}(X_n, \mathcal{L}(C))$. Complete function tables for the $\mathcal{L}_{\mathcal{T}}$ and $\mathcal{L}_{\mathcal{F}}$ operators are also shown in Fig. 16. Note that all three function tables are monotonic with respect to their inputs.

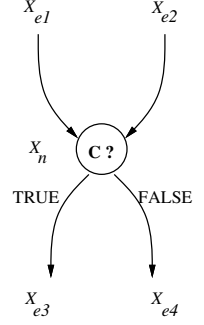
Other cases for mapping a node X_n value to the X_e values of its outgoing edges can be defined similarly. If n has exactly one outgoing edge e , then $X_e =$

Definition of join operator, $X_n = X_{e1} \sqcap X_{e2}$:

X_n	$X_{e2} = \text{NO}$	$X_{e2} = \text{MAYBE}$
$X_{e1} = \text{NO}$	NO	MAYBE
$X_{e1} = \text{MAYBE}$	MAYBE	MAYBE

**Definition of true operator for branch condition C ,
 $X_{e3} = \mathcal{L}_{\mathcal{T}}(X_n, \mathcal{L}(C))$:**

X_{e3}	$\mathcal{L}(C) = \top$	$\mathcal{L}(C) = \text{TRUE}$	$\mathcal{L}(C) = \text{FALSE}$	$\mathcal{L}(C) = \perp$
$X_n = \text{NO}$	NO	NO	NO	NO
$X_n = \text{MAYBE}$	NO	MAYBE	NO	MAYBE



**Definition of false operator for branch condition C ,
 $X_{e4} = \mathcal{L}_{\mathcal{F}}(X_n, \mathcal{L}(C))$:**

X_{e4}	$\mathcal{L}(C) = \top$	$\mathcal{L}(C) = \text{TRUE}$	$\mathcal{L}(C) = \text{FALSE}$	$\mathcal{L}(C) = \perp$
$X_n = \text{NO}$	NO	NO	NO	NO
$X_n = \text{MAYBE}$	NO	NO	MAYBE	MAYBE

Fig. 16. Executable flag mappings for join operator (\sqcap), true operator ($\mathcal{L}_{\mathcal{T}}$), and false operator ($\mathcal{L}_{\mathcal{F}}$)

$\mathcal{L}(k_3)$	$X_{e2} = \text{NO}$	$X_{e2} = \text{MAYBE}$
$X_{e1} = \text{NO}$	\top	$\mathcal{L}(k_2)$
$X_{e1} = \text{MAYBE}$	$\mathcal{L}(k_1)$	$\mathcal{L}(k_1) \sqcap \mathcal{L}(k_2)$

Fig. 17. $k_3 := \Phi(k_1, X_{e1}, k_2, X_{e2})$, where execution flags X_{e1} and X_{e2} control the selection of k_1 and k_2 respectively

X_n . If node n has more than two outgoing edges then the mapping for each edge is similar to the $\mathcal{L}_{\mathcal{T}}$ and $\mathcal{L}_{\mathcal{F}}$ operators.

Recall that the \mathcal{L}_{Φ} function for a control Φ was defined in Sect. 3.2 by the meet function, $\mathcal{L}_{\Phi}(\mathcal{L}(A_2), \mathcal{L}(A_1)) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_2)$. For the extended SCC algorithm described in this section, we use the definition of \mathcal{L}_{Φ} shown in Fig. 17. This definition uses executable flags X_{e1} and X_{e2} , where $e1$ and $e2$ are the incoming control flow edges for the Φ function. Thus, the lattice values of the executable flags X_{e1} and X_{e2} are used as compile-time approximations of @ variables $@k_1$ and $@k_2$.

4.2 Sparse Conditional Constant Propagation Algorithm

We introduce our algorithm by first explaining how it works for acyclic scalar code. Consider the example program shown in Fig. 18. The basic blocks are labeled $n1, n2, n3$ and $n4$. Edges $e12, e13, e24$ and $e34$ connect nodes in the obvious way. The control flow following block $n1$ depends on the value of variable n .

The first step is to transform the example in Fig. 18 to partial Array SSA form (with no @ variables) as shown in Fig. 19. Note that since k had multiple assignments in the original program, a ϕ function is required to compute k_3 as a function of k_1 and k_2 .

The second step is to use the partial Array SSA form to create a set of data flow equations on lattice values for use by our constant propagation algorithm. The conversion to equations is performed as follows. There is one equation created for each assignment statement in the program. There is one equation created for each node in the CFG. There is one equation created for each edge in the CFG. The equations for the assignments in our example are shown in Fig. 20. The equations for the nodes and edges in our example are found in Fig. 21 and 22 respectively.

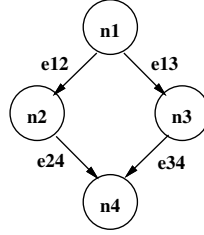
The lattice operations $\mathcal{L}_{<}$, \mathcal{L}_{*} , and \mathcal{L}_{max} use specific knowledge of their operation as well as the lattice values of their operands to compute resulting lattice operations. For example, $\mathcal{L}_{*}(\perp, \mathcal{L}(0))$ results in $\mathcal{L}(0)$ because the result of multiplying 0 by any number is 0.

Next, we employ a work list algorithm, shown in Fig. 23, that reevaluates the equations until there are no further changes. A solution to the data flow equations identifies lattice values for each variable in the Array SSA form of the program, and for each node executable flag and edge executable flag in the CFG. Reevaluation of an equation associated with an assignment may cause equations associated with other assignments to be inserted on the work list. If the value appears in a conditional expression, it may cause one of the equations associated with edges to be inserted on the work list. Reevaluation of an edge's executable flag may cause an equation for a destination node's executable flag to be inserted on the work list. If reevaluation of a node's executable flag indicates that the node may be evaluated, then the equations associated with assignments within that node to be added to the work list. When the algorithm terminates, the lattice values for variables identify the constants in the program, and the lattice values for executable flags of nodes identify unreachable code.

```

n1:  i := 1
      C := i < n
      if C then
n2:    k := 2 * i
      else
n3:    k := 2 * n
      endif
n4:  print k

```

**Fig. 18.** Acyclic Scalar Example

```

n1:  i := 1
      C := i < n
      if C then
n2:    k1 := 2 * i
      else
n3:    k2 := 2 * n
      endif
n4:  k3 := φ(k1, k2)
      print k3

```

Fig. 19. Partial Array SSA form for the Acyclic Scalar Example
$$\begin{aligned}
\mathcal{L}(i) &= \mathcal{L}(1) \\
\mathcal{L}(C) &= \mathcal{L}_{<}(\mathcal{L}(i), \mathcal{L}(n)) \\
\mathcal{L}(k_1) &= \mathcal{L}_*(\mathcal{L}(2), \mathcal{L}(i)) \\
\mathcal{L}(k_2) &= \mathcal{L}_*(\mathcal{L}(2), \mathcal{L}(n)) \\
\mathcal{L}(k_3) &= \mathcal{L}_\Phi(\mathcal{L}(k_1), X_{e24}, \mathcal{L}(k_2), X_{e34})
\end{aligned}$$
Fig. 20. Equations for Assignments
$$\begin{aligned}
X_{n1} &= \text{TRUE} \\
X_{n2} &= X_{e12} \\
X_{n3} &= X_{e13} \\
X_{n4} &= X_{e24} \sqcap X_{e34}
\end{aligned}$$
Fig. 21. Equations for Nodes

$$\begin{aligned}
X_{e12} &= \mathcal{L}_{\mathcal{T}}(X_{n1}, \mathcal{L}(C)) \\
X_{e13} &= \mathcal{L}_{\mathcal{F}}(X_{n1}, \mathcal{L}(C)) \\
X_{e24} &= X_{n2} \\
X_{e34} &= X_{n3}
\end{aligned}$$

Fig. 22. Equations for Edges**Initialization:**

```

 $\mathcal{L}(v) \leftarrow \top$  for all local variables,  $v$ .
 $X_n \leftarrow \text{MAYBE}$  where  $X_n$  is the executable flag for the entry node.
 $X_n \leftarrow \text{NO}$  where  $X_n$  is the executable flag for any node other than the entry node.
 $X_e \leftarrow \text{NO}$  where  $X_e$  is the executable flag for any edge.
insert( $E_v, \text{work\_list}$ ) for each equation  $E_v$  defining  $v$ 
  such that  $v$  is assigned to in the entry block.

```

Body:

```

while ( $\text{work\_list} \neq \text{empty}$ )
   $E_v \leftarrow \text{remove}(\text{work\_list})$ 
  reevaluate( $E_v$ )
  insert( $E'_{v'}, \text{work\_list}$ ) for each equation  $E'_{v'}$  that uses  $E_v$ 
  if  $E_v$  defines the executable flag for some node  $n$  then
    insert( $E'_{v'}, \text{work\_list}$ ) for each equation  $E'_{v'}$  defining  $v'$ 
    such that  $v'$  is assigned to in block  $n$ .
  end if
end while

```

Fig. 23. Sparse Conditional Constant propagation (SCC) algorithm for scalar and array variables

Even though we assumed an acyclic CFG in the above discussion, the algorithm in Fig. 23 can be used unchanged for performing constant propagation analysis on a CFG that may have cycles. The only difference is that the CFG may now contain back edges. Each back edge will be evaluated when its source node is modified. The evaluation of this back edge may result in the reevaluation of its target node.

As in past work, it is easy to show that the algorithm must take at most $O(Q)$ time, where Q is the number of data flow equations, assuming that the maximum arity of a function is constant and the maximum height of the lattice is constant.

As an example with array variables, Fig. 24 lists the data flow equations for the assignment statements in the Array SSA program in Fig. 6 (the data flow equations for nodes and edges follow the CFG structure as in Figs. 21 and 22). Given the definition of lattice elements for array variables from Sect. 3.1, the conditional constant propagation algorithm in Fig. 23 can also be used unchanged for array variables.

$$\begin{aligned}
\mathcal{L}(A_0) &= \perp \\
\mathcal{L}(i) &= \mathcal{L}(1) \\
\mathcal{L}(C) &= \mathcal{L}_{<}(\mathcal{L}(i), \mathcal{L}(n)) \\
\mathcal{L}(k) &= \mathcal{L}_*(\mathcal{L}(2), \mathcal{L}(i)) \\
\mathcal{L}(A_1) &= \mathcal{L}_{d[] }(\mathcal{L}(k), \mathcal{L}(i)) \\
\mathcal{L}(A_2) &= \mathcal{L}_{d\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0)) \\
\mathcal{L}(A_3) &= \mathcal{L}_{\Phi}(\mathcal{L}(A_2), X_{e24}, \mathcal{L}(A_0), X_{e14})
\end{aligned}$$

Fig. 24. Equations for Assignments from Fig. 16

5 Related Work

Static single assignment (SSA) form for scalar variables has been a significant advance. It has simplified the design of some optimizations and has made other optimizations more effective. The popularity of SSA form surged after an efficient algorithm for computing SSA form was made available [4]. SSA form is now a standard representation used in modern optimizing compilers in both industry and academia.

However, it has been widely recognized that SSA form is much less effective for array variables than for scalar variables. The approach recommended in [4] is to treat an entire array like a single scalar variable in SSA form. The most serious limitation of this approach is that it lacks precise data flow information on a per-element basis. Array SSA form addresses this limitation by providing Φ functions that can combine array values on a per-element basis. The constant propagation algorithm described in this paper can propagate lattice values through Φ functions in Array SSA form, just like any other operation/function in the input program.

The problem of conditional constant propagation for scalar variables has been studied for several years. Wegbreit [7] provided a general algorithm for solving data flow equations; his algorithm can be used to perform conditional constant propagation and more general combinations of program analyses. However, his algorithm was too slow to be practical for use on large programs. Wegman and Zadeck [8] introduced a Sparse Conditional Constant (SCC) propagation algorithm that is as precise as the conditional constant propagation obtained by Wegbreit’s algorithm, but runs faster than Wegbreit’s algorithm by a speedup factor that is at least $O(V)$, where V is the number of variables in the program. The improved efficiency of the SCC algorithm made it practical to perform conditional constant propagation on large programs, even in the context of industry-strength product compilers. The main limitation of the SCC algorithm is a conceptual one — the algorithm operates on two “worklists” (one containing edges in the SSA graph and another containing edges from the control flow

graph) rather than on data flow equations. The lack of data flow equations makes it hard to combine the algorithm in [8] with other program analyses. The problem of combining different program analyses based on scalar SSA form has been addressed by Click and Cooper in [2], where they present a framework for combining constant propagation, unreachable-code elimination, and value numbering that explicitly uses data flow equations.

Of the conditional constant propagation algorithms mentioned above, our work is most closely related to that of [2] with two significant differences. First, our algorithm performs conditional constant propagation through both scalar and array references, while the algorithm in [2] is limited only to scalar variables. Second, the framework in [2] uses control flow predicates instead of execution flags. It wasn't clear from the description in [2] how their framework deals with predicates that are logical combinations of multiple branch conditions; it appears that they must either allow the possibility of an arbitrary size predicate expression appearing in a data flow equation (which would increase the worst-case execution time complexity of their algorithm) or they must sacrifice precision by working with an approximation of the predicate expression.

6 Conclusions

We have presented a new sparse conditional constant propagation algorithm for scalar and array references based on Array SSA form [6]. Array SSA form has two advantages: It is designed to support analysis of arrays at the element level and it employs a new Φ function that is a pure function of its operands, and can be manipulated by the compiler just like any other operator in the input program.

The original sparse conditional constant propagation algorithm in [8] dealt with control flow and data flow separately by maintaining two distinct work lists. Our algorithm uses a single set of data flow equations and is therefore conceptually simpler. In addition to being simpler, the algorithm presented in this paper is more powerful than its predecessors in that it handles constant propagation through array elements. It is also more effective because its use of data flow equations allows it to be totally integrated with other data flow algorithms, thus making it easier to combine other analyses with conditional constant propagation.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. Cliff Click and Keith D. Cooper. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.
3. The Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. <http://open.specbench.org/osg/cpu95/>, 1997.

4. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
5. Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, Inc., 1977.
6. Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in Parallelization. *Conf. Rec. Twenty-fifth ACM Symposium on Principles of Programming Languages, San Diego, California*, January 1998.
7. B. Wegbreit. Property Extraction in Well-Founded Property Sets. *IEEE Transactions on Software Engineering*, 1:270–285, 1975.
8. Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses [★]

Michael Hind and Anthony Pioli

State University of New York at New Paltz
IBM Thomas J. Watson Research Center
{hind,pioli}@mcs.newpaltz.edu

Abstract. This paper describes an empirical comparison of four context-insensitive pointer alias analysis algorithms that use varying degrees of flow-sensitivity: a flow-insensitive algorithm that tracks variables whose addresses were taken and stored; a flow-insensitive algorithm that computes a solution for each function; a variant of this algorithm that uses precomputed kill information; and a flow-sensitive algorithm. In addition to contrasting the precision and efficiency of these analyses, we describe implementation techniques and quantify their analysis-time speed-up.

1 Introduction

To effectively analyze programs written in languages that make extensive use of pointers, such as C, C++, or Java (in the form of references), knowledge of pointer behavior is required. Without such knowledge, conservative assumptions about pointer values must be made, resulting in less precise data flow information, which can adversely affect the effectiveness of analyses and tools that depend on this information.

A pointer alias analysis is a compile-time analysis that, for each program point, attempts to determine what a pointer can point to. As such an analysis is, in general, undecidable [25, 34], approximation methods have been developed. These algorithms provide trade-offs between the efficiency of the analysis and the precision of the computed solution. The goal of this work is to quantify how the use of flow-sensitivity affects precision and efficiency.

Although several researchers have provided empirical results of their techniques, comparisons among algorithms can be difficult because of differing program representations, benchmark suites, and metrics. By holding these factors constant, we can focus more on the efficacy of the algorithms and less on the manner in which the results were obtained.

The contributions of this paper are the following:

- empirical results that measure the precision and efficiency of four pointer alias analysis algorithms with varying degrees of flow-sensitivity;

[★] This work was supported in part by the National Science Foundation under grant CCR-9633010, by IBM Research, and by SUNY at New Paltz Research and Creative Project Awards.

- empirical evidence of how various implementation enhancements significantly improved analysis time of the flow-sensitive analysis.

In addition to the use of flow-sensitivity, other factors that affect the cost/precision trade-offs of pointer alias analyses include the use of context-sensitivity and the manner in which aggregates (arrays and structs) and the heap are modeled. Our experiments hold these factors constant so that the results only reflect the usage of flow-sensitivity.

Section 2 highlights the four algorithms and their implementations. Section 3 describes the empirical study of the four algorithms, analyzes the results, and contrasts them with related results from other researchers. Section 4 overviews some of the performance-improving enhancements made in the implementation and quantifies their analysis-time speed-up. Section 5 describes other related work. Section 6 states conclusions.

2 Analyses and Implementation

One manner of classifying interprocedural data flow analyses is whether they consider control flow information during the analysis. A *flow-sensitive* analysis considers control flow information of a procedure during its analysis of the procedure. A *flow-insensitive* analysis does not consider control flow information during its analysis, and thus can be more efficient, but less precise. (See [31] for a full discussion of these definitions.)

The algorithms we consider, listed in order of increasing precision, are

- AT:** a flow-insensitive algorithm that computes one solution set for the entire program that contains all named objects whose address has been taken and stored,
- FI:** a flow-insensitive algorithm [4, 5] that computes a solution set for every function,
- FIK:** a flow-insensitive algorithm [4, 5] that computes a solution set for every function, but attempts to improve precision by using precomputed (flow-sensitive) kill information,
- FS:** a flow-sensitive algorithm [8, 5] that computes a solution set for every program point.

The following sections provide further information about these analyses and their implementation.

2.1 Algorithms

The program is represented as a program call (multi-) graph (PCG), in which a node corresponds to a function, and a directed edge represents a call to the target function.¹ Each function body is represented by a control flow graph (CFG). This

¹ Potential calls can occur due to function pointers and virtual methods, in which the called function is not known until run time.

graph is used to build a simplified sparse evaluation graph (SEG) [9], discussed in Section 4.

The address-taken analysis (AT) computes its solution by making a single pass over all functions in the program, adding to a global set all variables whose addresses have been assigned to another variable. These include actual parameters whose addresses are stored in the corresponding formal. Examples are statements such as “`p = &a;`”, “`q = new ...;`”, and “`foo(&a);`”, but not simple expression statements such as “`&a;`” because the address was not stored. AT is efficient because it is linear in the size of the program and uses a single solution set, but it can be very imprecise. It is provided as a base case for comparison to the other three algorithms presented in this paper.

The general manner in which the other three analyses compute their solutions is the same. A nested fixed point computation is used in which the outer nest corresponds to computing solutions for each function in the PCG. Each such function computation triggers the computation of a local solution for all program points that are distinguished in the particular analysis. For the flow-sensitive (FS) analysis, the local solution corresponds to each CFG node in the function. For the other two flow-insensitive analyses (FI, FIK), the local solution corresponds to one set that conservatively represents what can hold anywhere in the function. This general framework is presented as an iterative algorithm in Fig. 1 and is further described in [5]. An extension to handle virtual methods is described in [6]. Section 4 reports improvements due to the use of a worklist-based implementation.

```

S1:   build the initial PCG
S2:   foreach procedure, p, in the PCG, loop
S3:       initialize interprocedural alias sets of p to {}
S4:   end loop
S5:   repeat
S6:       foreach procedure, p, in the PCG, loop
S7:           using the interprocedural alias sets (for entry of p and call sites in p),
                compute the intraprocedural alias sets of p
S8:           using the intraprocedural alias sets of p,
                update the interprocedural alias sets representing
                the effect of p on each procedure that calls or is called by p
S9:       end loop
S10:   using new function pointer aliases, update the PCG,
                initializing interprocedural alias sets of new functions to {}
S11: until the interprocedural alias sets and the PCG converge

```

Fig. 1. High-level description of general algorithm [5]

² As the PCG is not used, this can include functions that are not called.

The FS, FI, and FIK analyses utilize the *compact representation* [8, 5] to represent alias relations. This representation shares the property of the *points-to* representation [14], in that it captures the “edge” information of alias relations. For example, if variable **a** points to **b**, which in turn points to **c**, the compact representation records only the following alias set: $\{\langle *a, b \rangle, \langle *b, c \rangle\}$, from which it can be inferred that $\langle **a, c \rangle$ and $\langle **a, *b \rangle$ are also aliases.³

All analyses are *context-insensitive*; they merge information flowing from different calls to the same function, and may suffer from the *unrealizable path problem* [27], i.e., they potentially propagate back to the wrong caller the aliases of the called function. (Sections 3.2 and 4.4 discuss this potential imprecision.) Context-sensitive analyses [14, 48] do not suffer from this problem, but may increase time/space costs.

As in [22, 7], all analyses considered here represent the (possibly many) objects allocated at calls to **new** or **malloc** by creating a named object based on the CFG node number of the allocation statement. These objects are referred to as $heap_n$, where n is the CFG node number of the allocation statement. These names are unique throughout the entire program. More precise heap modeling schemes [29, 22, 19, 7, 8, 11, 15, 16, 39, 40] can improve precision, but may also increase time/space costs. Quantifying the effects of using context-sensitivity and various heap models is beyond the scope of this work.

Consider the simple program in Fig. 2. The AT analysis computes only one set of objects, which it assumes all pointers may point to. This set will contain five objects $\{heap_{S1}, heap_{S3}, heap_{S4}, heap_{S6}, \text{ and } heap_{S7}\}$, all of which will appear to be referenced at $S8$.

<pre> T *p, *q; void main(void) { S1: p = new T; S2: f(); S3: p = new T; } </pre>	<pre> void f(void) { S4: p = new T; S5: g(); S6: p = new T; S7: q = new T; } </pre>	<pre> void g(void) { S8: T t = *p; } </pre>
---	---	---

Fig. 2. Example program

The FI analysis does not use any intraprocedural control flow information. Instead it conservatively computes what can hold anywhere within a function, i.e., for each function, f , it uses only one alias set, $Holds_f$, to represent what may hold at any CFG node in f . Thus, in Fig. 2 the FI analysis assumes that $\langle *p, heap_{S1} \rangle$ and $\langle *p, heap_{S3} \rangle$ can flow into **f**. This results in $Holds_g = Holds_f = \{\langle *p, heap_{S1} \rangle, \langle *p, heap_{S3} \rangle, \langle *p, heap_{S4} \rangle, \langle *p, heap_{S6} \rangle, \langle *q, heap_{S7} \rangle\}$, which re-

³ See [30, 5] for a discussion of precision trade-offs between this representation and an explicit representation, which would contain all four alias pairs.

sults in four objects, $heap_{S1}$, $heap_{S3}$, $heap_{S4}$, and $heap_{S6}$ potentially being referenced at $S8$.⁴

The FIK analysis attempts to improve the precision of the flow-insensitive analysis by *precomputing* kill information for pointers, and then uses this information during the flow-insensitive analysis at call sites.⁵ For example, the precomputation will determine that all alias relations involving $*p$ on entry to f will be killed before the call to g at $S5$. Thus, $Holds_g$ will contain only the alias relations that are generated in f and propagated to g , i.e., $Holds_g = \{\langle *p, heap_{S4} \rangle, \langle *p, heap_{S6} \rangle \langle *q, heap_{S7} \rangle\}$. This results in two objects, $heap_{S4}$ and $heap_{S6}$, potentially being referenced at $S8$.

The FS analysis associates an alias set before (In_n) and after (Out_n) every CFG node, n . For example, $Out_{S1} = \{\langle *p, heap_{S1} \rangle\}$ because $*p$ and $heap_{S1}$ refer to the same storage. At the entry to function g , the FS analysis will compute $In_{S8} = \{\langle *p, heap_{S4} \rangle\}$, which is the precise solution for this simple example.

This example illustrates the theoretical precision levels of the four analyses, from FS (most precise) to AT (least precise). The AT analysis is our most efficient analysis because it is linear and only uses one set. The FI analysis is more efficient than the FIK analysis because it neither precomputes kill information nor uses it during the analysis. One would expect the FS analysis to be the least efficient because it needs to distinguish solutions for every point in the program. Thus, a theoretical spectrum exists in terms of precision and efficiency with the AT analysis on the less precise/more efficient side, the FS analysis on the more precise/less efficient side, and the FI and FIK analyses in the middle.

Not studied in this paper are other flow-insensitive analyses [45, 42] that use one alias set for the whole program and limit the number of alias relations by sometimes grouping distinct variables into one named object. These analysis fall in between AT and FI in the theoretical precision spectrum.

2.2 Implementation

The analyses have been implemented in the NPIC system, an experimental program analysis system written in C++. The system uses multiple and virtual inheritance to provide an extensible framework for data flow analyses [21, 33]. A prototype version of the IBM VisualAge C++ compiler [43, 32] is used as the front end. The abstract syntax tree constructed by the front end is transformed into a PCG and a CFG for each function, which serve as input to the alias analyses. No CFG is built for library functions. We model a call to a library function based on its semantics, thereby providing the benefits of context-sensitive analysis of such calls. Library calls that cannot affect the value of a pointer are treated as the identity transfer function.

⁴ Although a final intraprocedural flow-sensitive pass can be used to improve precision [5], this pass has not been implemented.

⁵ Kill information is computed in a single flow-sensitive prepass of each CFG. For each call site, c , we compute two sets, the set of pointers that are definitely killed on all paths from entry to c and the set of pointers that are definitely killed on all paths from c to exit [4, 5]. Only the first set is used in our example.

The FS, FI, and FIK analyses are implemented using worklists. (Section 4.2 discusses an earlier iterative implementation.) These three analyses incorporate function pointer analysis into the pointer alias analysis as described in [4, 5]. Currently, array elements and field components are not distinguished, and `setjmp/longjmp` statements are not supported. The implementation also assumes that pointer values will only exist in pointer variables, and that pointer arithmetic does not result in the pointer going beyond array boundaries. As stated in Section 2.1, heap objects are named based on their allocation site.

To model the values passed as `argc` and `argv` to the `main` function, a dummy `main` function was added, which called the benchmark’s `main` function, simulating the effects of `argc` and `argv`. This function also initialized the `_iob` array, used for standard I/O. The added function is similar to the one added by Ruf [36, 38] and Landi et al. [28, 26]. Initializations of global variables are automatically modeled as assignment statements in the dummy `main` function.

3 Results

Our benchmark suite contains 21 C programs, 18 provided by other researchers [28, 14, 36] and 3 from the SPEC CINT92 [3] and CINT95 [44] benchmarks.⁷ Table 1 describes characteristics of the suite. The third column contains the number of lines in the source and header files reported by the Unix utility `wc`. The fourth column reports the number of user-defined functions (nodes in the PCG), which include the dummy `main` function. The next two columns give the number of call sites, distinguished between user and library function calls. The next two columns report cumulative statistics for all CFG nodes and edges. These figures include nodes and edges created by the initialization of globals. The following column computes the ratio of CFG edges to nodes. The next column reports the percentage of CFG nodes that are considered pointer-assignment nodes. The current analysis treats an assignment as a pointer-assignment if the variable involved in the pointer expression on the left side of the assignment is declared to be a pointer.⁸ The last two columns report the number of recursive functions (functions that are in PCG cycles) and heap allocation sites in each program. The last row of the table reports the average edge/node ratio and the

⁶ Although one program in our benchmark suite, `anagram`, does syntactically contain a call to `longjmp`, the code is unreachable.

⁷ Some programs had to be syntactically modified to satisfy C++’s stricter type checking semantics. A few program names are different than those reported by Ruf [36]. The SPEC CINT92 program 052.alvinn was named `backprop` in Todd Austin’s benchmark suite [2]. Ruf referred to `ks` as `part`, and `ft` as `span` [38].

⁸ This is more conservative than considering statements in which the left side *expression* is a pointer. Thus, statements such as “`p->field = ...`” are treated as pointer assignments no matter how the type of `field` is declared. A more accurate categorization would not affect the precision of the analysis, but could improve the efficiency by reducing the number of nodes considered during the analysis as discussed in Section 4.1

average pointer-assignment node percentage, both of which are computed by averaging the corresponding values over the 21 benchmarks.

Table 1. Static characteristics of benchmark suite

Name	Source	LOC	Funcs	Call Sites		CFG			Ptr-Asgn Nodes	Rec Funcs	Allocation Sites
				User	Lib	Nodes	Edges	$\frac{Edges}{Nodes}$			
allroots	Landi	227	7	19	35	157	167	1.064	2.6%	2	1
052.alvinn	SPEC92	272	9	8	13	223	243	1.090	11.2%	0	0
01.qbsort	McCat	325	8	9	25	173	187	1.081	24.9%	1	5
15.trie	McCat	358	13	19	21	167	181	1.084	23.4%	3	5
04.bisect	McCat	463	9	11	18	175	189	1.080	9.7%	0	2
17.bintr	McCat	496	17	27	28	196	205	1.046	9.7%	5	1
anagram	Austin	650	16	22	38	332	365	1.099	9.0%	1	2
lex315	Landi	733	17	102	52	527	615	1.167	2.5%	0	3
ks	Austin	782	14	17	67	513	576	1.123	26.9%	0	5
05.eks	McCat	1,202	30	62	49	678	732	1.080	3.8%	0	3
08.main	McCat	1,206	41	68	53	684	700	1.023	24.3%	3	8
09.vor	McCat	1,406	52	174	28	876	917	1.047	27.5%	5	8
loader	Landi	1,539	30	79	102	687	773	1.125	8.3%	2	7
129.compress	SPEC95	1,934	25	35	28	596	652	1.094	6.4%	0	0
ft	Austin	2,156	38	63	55	732	808	1.104	19.5%	0	5
football	Landi	2,354	58	257	274	2,710	3,164	1.168	1.8%	1	0
compiler	Landi	2,360	40	349	107	1,723	2,090	1.213	1.1%	14	0
assembler	Landi	3,446	52	247	243	1,544	1,738	1.126	8.0%	0	16
yacr2	Austin	3,979	59	158	169	2,030	2,328	1.147	5.4%	5	26
simulator	Landi	4,639	111	447	226	2,686	3,123	1.163	2.7%	0	4
099.go	SPEC95	29,637	373	2,054	22	16,823	20,176	1.199	.2%	1	0
Average								1.111	10.9%		

3.1 Description of Experiment

This section presents precision and efficiency results. For each benchmark and each analysis, we report the analysis time, the maximum memory used, and the average number of objects the analysis computes a dereferenced pointer can point to. The precision results for the FIK analysis are exactly the same as the FI analysis for all benchmarks. Thus, we do not explicitly include this analysis in our precision data.

The experiment was performed on a 132MHz IBM RS/6000 PowerPC 604 with 96MB RAM and 104MB paging space, running AIX 4.1.5. The executable was built with IBM’s xlc compiler using the “-O3” option.

The analysis time is reported in seconds and does not include the time to build the PCG and CFGs, but does include any analysis-specific preprocessing, such as building the SEG from the CFG. This information is displayed in the top left chart in Fig. 3. The top right chart of this figure reports the high-water mark of memory usage during the analysis process. This memory usage includes the intermediate representation, the alias information, and statistics-related data. This information was obtained by using the “ps v” command under AIX 4.1.5.

To collect precision information, the system traverses the representation visiting each expression containing a pointer dereference and, using the computed

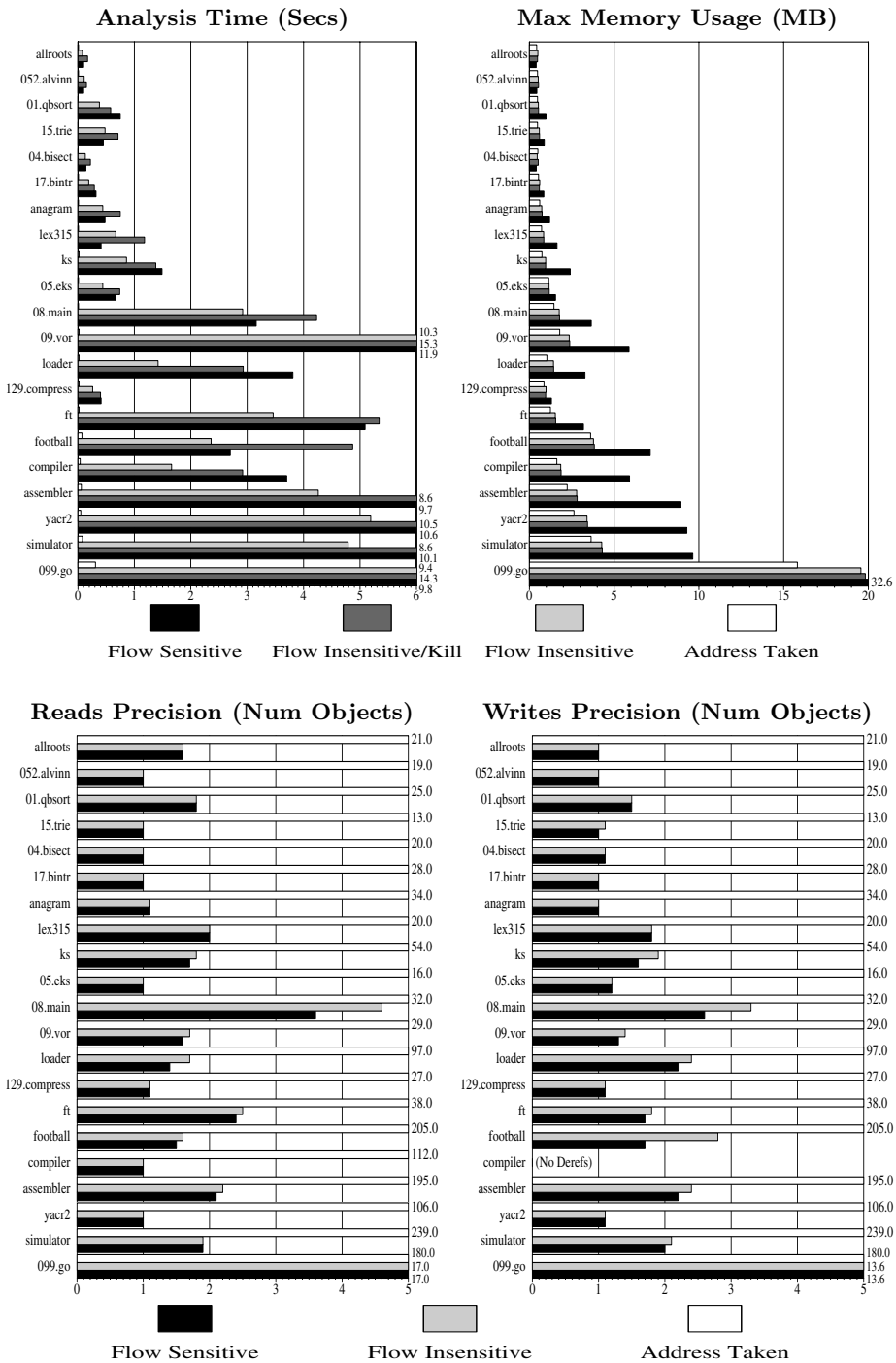


Fig. 3. Analysis time, memory usage, and precision results

alias information, reports how many named objects are aliased to the pointer expression. We report the average number of such dereferences for both reads and writes. Further precision information is provided in [21].

A pointer expression with multiple dereferences, such as `***p`, is counted as multiple dereference expressions, one for each dereference. The intermediate dereferences (`*p` and `**p`) are counted as reads. The last dereference (`***p`) is counted as a read or write depending on the context of the expression. Statements such as `(*p)++` and `*p += increment` are treated as both a read and a write of `*p`.

We consider a pointer to be dereferenced if the variable is declared as a pointer or an array formal parameter, and one or more of the “`*`”, “`->`”, or “`[]`” operators are used with that variable. Formal parameter arrays are included because their corresponding actual parameter(s) could be a pointer. We do not count the use of the “`[]`” operator on arrays that are not formal parameters because the resulting “pointer” (the array name) is constant, and therefore, counting it may skew results. Figure 4 classifies the type of pointer dereferenced averaged over all programs. Information for each benchmark is given in [21].

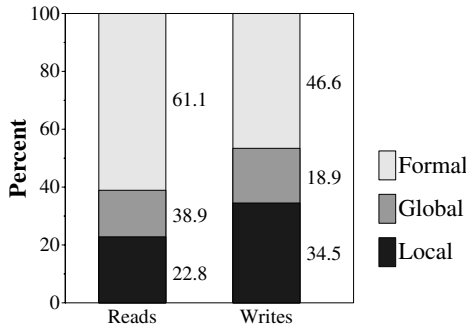
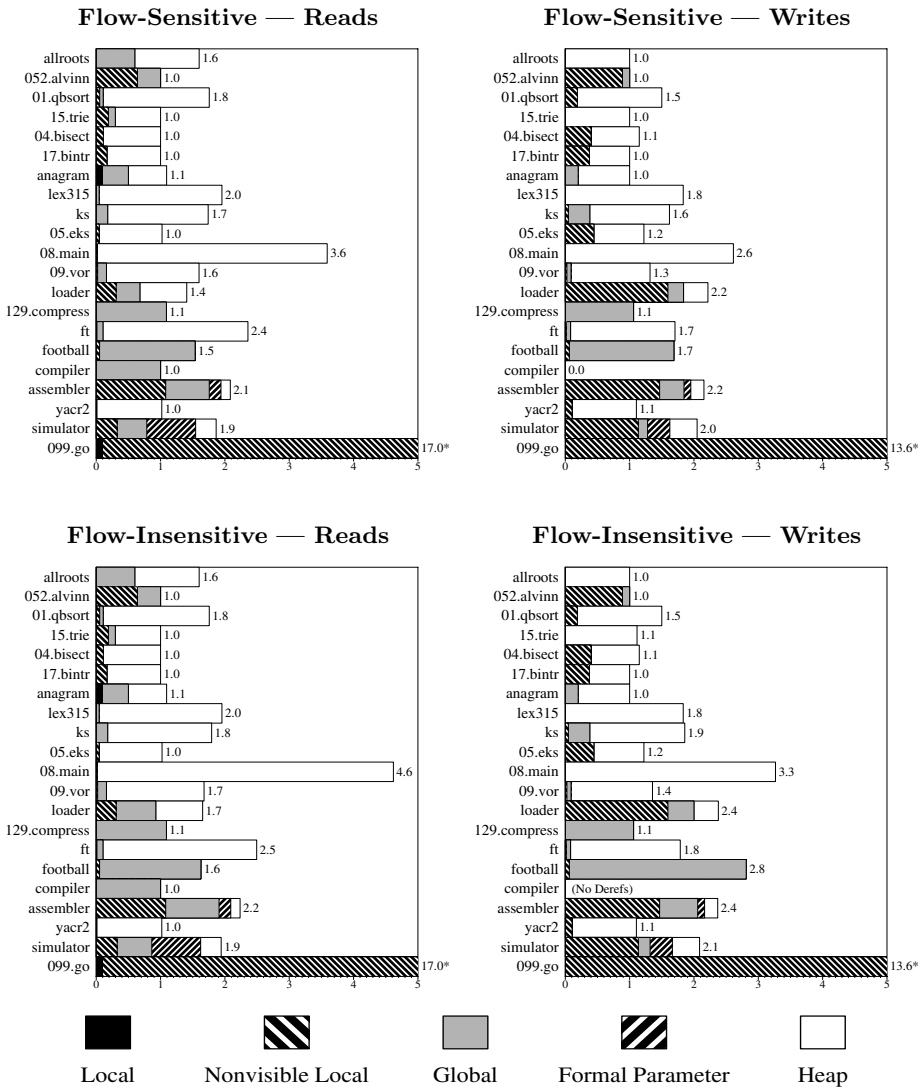


Fig. 4. Classification of dereferenced pointer types for all programs

The manner in which the heap is modeled must be considered in evaluating precision results. For example, a model that uses several names for objects in the heap may seem less precise when compared to a model that uses fewer names [36]. Similarly, analyses that represent invisible objects (objects not lexically visible in the current procedure) aliased to a formal parameter as a single object [9] may report fewer objects. Our analyses do not use this technique.

Assuming a correct input program, each pointer dereference should correspond to at least one object at run time, and thus one serves as a lower bound for the average. Although a precision result close to one demonstrates the analysis is precise (modulo heap and invisible object naming), a larger number could reflect an imprecise algorithm, a limitation of static analysis, or a pointer dereference that corresponds to different memory locations over the program’s execution.

⁹ This modeling technique can increase the possibility of strong updates [7].



* The bars for 099.go are truncated. The object type breakdown for both FS and FI is

	Nonvisible		Formal		
	Local	Local	Global	Parameter	Heap
Reads	0.1	9.7	7.2	0.01	0
Writes	0.0	8.1	5.4	0.01	0

Fig. 5. Breakdown of average object type pointed to by a dereferenced pointer

The bottom two charts of Fig. 3 provide a graphical layout of precision information for reads and writes. Fig. 5 refines this information for the FS and FI analyses by providing a breakdown of the type of object pointed to. Fig. 6 provides similar information averaged over all programs. Charts E and F of this figure report the percentage of dereferenced pointers that resolve to exactly one object in our model. If the object is a named variable, as opposed to a heap object, the pointer dereference could be replaced with the variable. This information is expanded upon in [21].

3.2 Discussion

As expected, the results from the analysis speed chart of Fig. 3 indicate that the AT analysis is efficient; it takes less than .4 seconds on all programs. The FI analysis can be over twice as fast as the FS analysis, and was faster than the FS analysis in all but one program.

The precision of the AT analysis leaves much to be desired. Fig. 6 reports on average 111.9 objects for reads and 96.68 objects for writes were in the AT set.¹⁰ As one would expect this set to increase with the size of the program, the precision for this analysis will worsen with larger programs.

The results also indicate that the FIK analysis is not beneficial. On our benchmark suite it is never more precise than the FI analysis, and on some occasions requires more analysis time than the FS analysis. One explanation for the precision result may be that an alias relation created to simulate a reference parameter, in which the formal points to the actual, typically is not killed in the called routine, i.e., the formal parameter is not modified, but rather is used to access the passed actual. Thus, programs containing these alias relations will not benefit from the precomputed kill information.

One surprising result is the overall precision of the FI analysis. In 12 of the 21 benchmarks the FI analysis is as precise as the FS analysis. This seems to suggest that the added precision obtained by the FS analysis in considering control flow within a function is not significant for those benchmarks, at least where pointers are dereferenced. We offer two possible explanations:

1. Pointer variables are often not assigned more than one distinguished object within the same function. Thus, distinguishing program points within a function, a key difference between the FS and FI analyses, does not often result in an increase in precision. We have seen exceptions to this in the function `InitLists` of the `ks` benchmark and in the function `InsertPoint` in the `08.main` benchmark. In both cases the same pointer is reused in two list-traversing loops.
2. It seems that a large number of alias relations are created at call sites because of actual/formal parameter bindings. The lack of a substantial precision difference between our FS and FI analyses may be because both algorithms rely on the same (context-insensitive) mapping mechanism at call sites.

¹⁰ The numbers differ because they are weighted with the number of reads and writes through a pointer in each program.

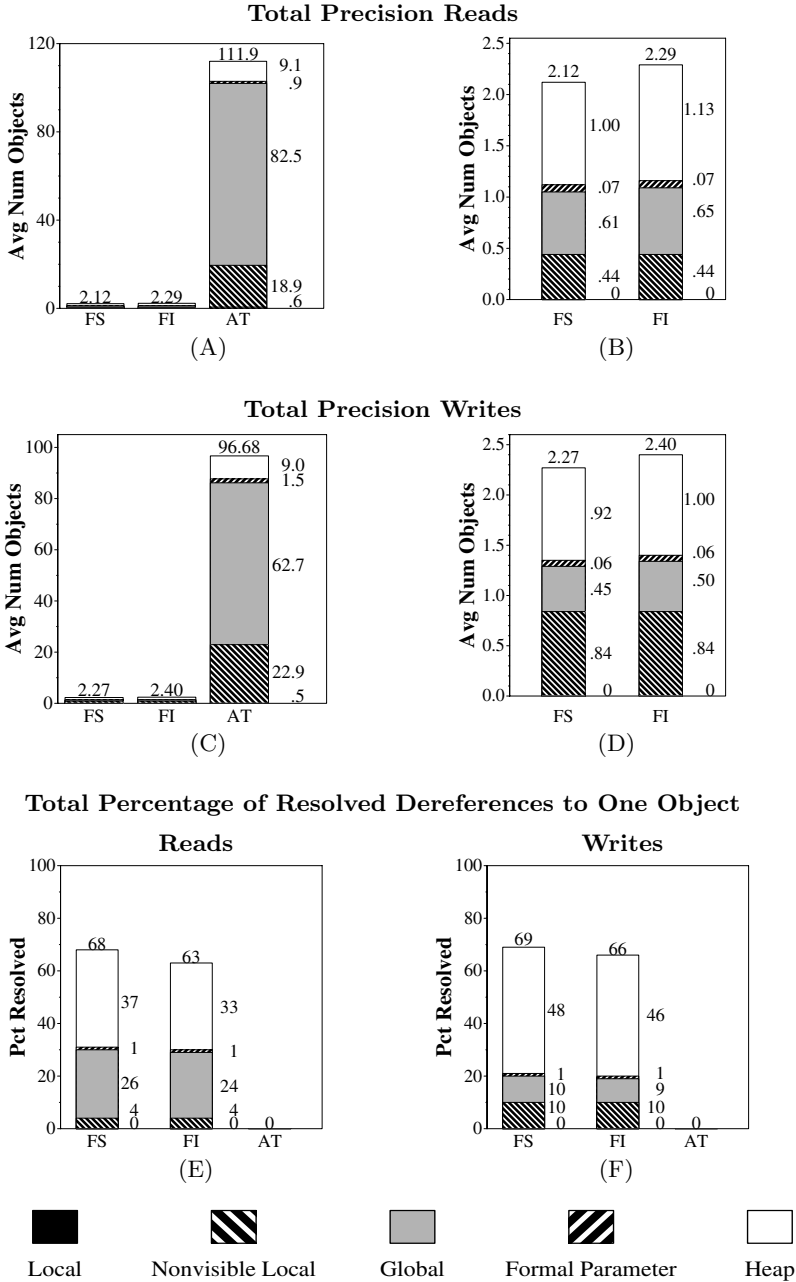


Fig. 6. Charts A – D provide the average precision over all benchmarks for reads and writes. (Charts B and D do not include the AT analysis to allow the difference between the FS and FI analyses to be visible.) Charts E and F report the percentage of dereferenced pointers that resolve to one object in our model.

Considering charts B and D of Fig. 3, it seems that FI is as precise as FS for pointers directed to nonvisible locals and formal parameters. Therefore, FS, if employed at all, should focus on pointers directed to the heap and global variables.

The precision results for `099.go` merit discussion. An average of 17.03 and 13.64 objects are returned for reads and writes, respectively, with a maximum of 100. This program contains six small list-processing functions (using an array-based “cursor” implementation) that accept a pointer to the head of a list as a parameter. One of these functions, `addlist`, is called 404 times, passing the address of 100 different actuals for the list header, resulting in 100 aliases for the formal parameter. However, because the lifetime of the formal is limited to this function (it does not call any other function), these relations are not propagated to any other function. Therefore, these relations do not suffer the effects of the unrealizable path problem mentioned in Section 2.1.

Another conclusion from the results is that analysis time is not only a function of program size; it also depends on the amount of alias relation propagation along the PCG and SEGs. For example, `099.go`, despite being our largest program and having a pointer aliased with 100 objects, is analyzed at one of the fastest rates (3,037 LOC/second, 1,724 CFG nodes/second) because these relations are not propagated throughout the program.

As suggested by Shapiro and Horwitz [41] and Diwan [12], a more precise and time-consuming alias analysis may not be as inefficient as it may appear because the time required to obtain increased precision may reduce the time required by subsequent analyses that utilize mod-use information, and thus pointer alias information, as their input. As the previous paragraph suggests, this can also be true about pointer alias analysis itself, which also utilizes pointer alias information during its analysis.

3.3 Comparison with Other Results

Landi et al. [28] report precision results for the computation of the MOD problem using a flow-sensitive pointer alias algorithm with limited context-sensitive information. Among the metrics they report is the number of “thru-deref” assigns, which corresponds to the “write” metrics reported in Fig. 3. However, since their results include compiler-introduced temporaries in their “thru-deref” count [26], a direct comparison is not possible.

Stocks et al. [46] use the same metric without including temporaries for the flow-sensitive context-sensitive analysis of Landi and Ryder [27]. They report the average number of objects ranges from 1.0 to 2.0 on the eight common benchmarks. On these benchmarks our flow-sensitive context-insensitive analysis ranges from 1.0 to 2.22. Two possible explanations for the slightly less precise results are 1) their algorithm uses some context-sensitivity; 2) the underlying representation is not identical, and thus pointer dereferences may not be counted in the same manner in all cases. For example, statements such as `cfree(TP)` located in `allroots` are treated as modifying the structure deallocated, and thus as a pointer dereference [26]. In fact, on the three programs in which our analysis

reports the same, or close to the same, number of “writes” as “thru-derefs” (`allroots`, `lex315`, `simulator`), our precision is identical to that reported in [46].

The relative precision of the flow-insensitive analysis compared to the flow-sensitive analysis is in contrast to the study of Stocks et al. [46], which compares the flow-sensitive analysis of Landi and Ryder [27] with a flow-insensitive analysis described in [49]. For the eight common benchmarks, our flow-insensitive algorithm ranges from 1.0 to 2.81 objects on average for a write dereference, compared to 1.0 to approximately 6.3 for the flow-insensitive analysis they studied. The analysis described in [49] shares the property of Steensgaard’s analysis [45] in that it groups all objects pointed-to by a variable into an equivalence class. Although this can lead to spurious alias relations not present in the FI analysis, it does allow for an almost linear time algorithm, which has been shown to be fast in practice [45, 42, 50].

Emami et al. [14] report precision results for a flow-sensitive context-sensitive algorithm. Their results range from 1.0 to 1.77 for all indirect accesses using a heap naming scheme that represents all heap objects with one name. Because we were unable to obtain the benchmarks from their suite, a direct comparison with our results is not possible.

Ruf [36] reports both read and write totals for a flow-sensitive context-insensitive analysis. However, unlike our analysis he counts use of the “[]” operator on arrays that are not formal parameters as a dereference [38]. Since such an array will always point to the same place, the average number of objects is improved.¹¹ For the 11 benchmarks in common,¹² Ruf reports an overall read and write average of 1.33 and 1.38, respectively. To facilitate comparisons, we have also counted in this manner. The results for the common benchmarks are averages of 1.35 and 1.47 for the FS analysis and 1.41 and 1.54 for the FI analysis. We attribute the slight differences in the FS analysis to the difference in representations. As Ruf [36] states, “the VDG intermediate representation often coalesces series of structure or array operations into a single memory write.” This coalescing can skew results in either direction.

Shapiro and Horwitz [41] present an empirical comparison of four flow-insensitive algorithms. The first algorithm is the same as the AT algorithm. The remaining three algorithms [45, 42, 1] can be less precise and more efficient than the algorithms studied in this paper.¹³ The authors measure the precision

¹¹ The best illustration of this is in `099.go`, which has a large number of array references, but a low number of pointer dereferences. In this program, the average changed from 17.03 to 1.13 for reads and from 13.64 to 1.48 for writes when all uses of the “[]” operator were counted.

¹² Although [36] reports results for `ft` (under the name `span`), our version of the benchmark is substantially larger than the one Ruf analyzed, and thus is not comparable.

¹³ In theory, the FI analysis can be more precise than Andersen’s algorithm [1] because it considers function scopes, at the cost of using more than one alias set. However, both algorithms are likely to offer similar precision in practice because the distinguishing case is probably uncommon.

of these analyses by implementing three dataflow analyses and an interprocedural slicing algorithm. In addition to these alias analysis clients, the authors also report the direct precision of the alias analysis algorithms in terms of the total number of points-to relations. We agree with [14, 36] that a more meaningful metric is to measure where the points-to information is used, such as where a pointer is dereferenced. They conclude 1) a more precise flow-insensitive analysis generally leads to increased precision by the subsequent analyses that use this information with varying magnitudes; 2) metrics measuring the alias analysis precision tend to be good predictors on the precision of subsequent analyses that use alias information; and 3) more precise flow-insensitive analysis can also improve the efficiency of subsequent analyses that use this information.

4 Efficiency Improvements

This section describes some of the performance-improving enhancements made in the implementation and quantifies their effects on analysis-time speed-up for the flow-sensitive algorithm. Although novelty is not claimed, the efficacy of each technique is shown.

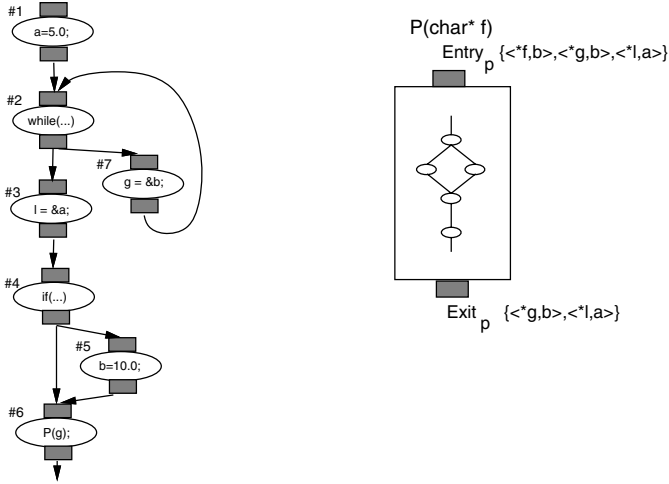


Fig. 7. Example CFG (left) and procedure P (right). Note that g is a global, a, b, l are locals of the CFG, and f is a formal parameter of P. Possible topological numbers (ignoring the back edge $7 \rightarrow 2$) appear next to each node.

4.1 Sharing Alias Sets

As described in Section 2.1, the flow-sensitive analysis computes solutions before and after every node in the CFG. However, this can result in storing redundant

information. For example, node #1 in Fig. 7 does not affect any pointer value; therefore its *Out* set will always equal its *In* set. Likewise, a node whose predecessor(s) *Out* set(s) have the same value will have an *In* set equal to this value. For example, all nodes in Fig. 7 except #1 and #2. Thus, for all nodes, except #1 and #2, alias sets can be shared. We use the term *shared* in a literal way — if the *In* set is shared with the *Out* set, they are the *same object* during the analysis. This is done with a shallow copy of the alias set data structures. We precompute these sharing sites in a separate forward pass over the CFG before performing the alias analysis. Each node that has its own set, which we call a *deep* set, is dubbed a “SEG” (*Sparse Evaluation Graph*) node.¹⁴ Such nodes have a list of “SEG” predecessors and “SEG” successors that are used during the analysis. The alias set allocation strategy for a node, n , is summarized as follows:

$$In_n = \begin{cases} Out_p, & \text{if } \forall p, q \in Pred(n), p \text{ and } q \text{ share } Out \text{ sets} \\ \text{a deep set,} & \text{otherwise} \end{cases}$$

$$Out_n = \begin{cases} In_n, & \text{if } n\text{'s transfer function is the identity function} \\ \text{a deep set,} & \text{otherwise} \end{cases}$$

Our current implementation treats all call nodes as SEG nodes.

Table 2 reports the number of alias sets before and after this optimization was applied as well as the percentage reduction. In addition to saving storage (on average 73.78% fewer alias sets were allocated), this method saves visits during the actual analysis to nodes that can not affect pointer aliasing. Although we allocate fewer alias sets, the real benefit of this technique is seen during the analysis. Since we do not visit and propagate alias relations between extraneous CFG nodes, we simply have fewer alias relations being stored and copied from one CFG node to the next. This affects both the analysis’s time and space use in a significant way.

Table 3 shows the effects of the efficiency improving ideas on the flow-sensitive analysis. Run times were collected as in Section 3 using the C function `clock()`, which gives the CPU time, not the elapsed time. This metric was chosen because it eliminates the effects of system load, amount of RAM vs. paging space, etc. The elapsed time for each program was approximately 2.5 times the CPU time.

The headings for each column are read vertically — for example, the second column shows the analysis time in seconds with no sharing of alias sets as well as no other enhancements described in this section. The third column reports the analysis time and resulting speed-up using this sharing technique. The effectiveness of this technique is mostly related to the percentage of CFG nodes that affect a pointer and percentage of call nodes — the higher the percentages the lower the potential benefit.¹⁵ For all our benchmarks, these averages were 10.9%

¹⁴ The method described is a conservative approximation to the SEG of [9] and is similar to [35].

¹⁵ The percentage of merge nodes plays a smaller role.

Table 2. Effectiveness of sharing alias sets

Benchmark	Num. Alias Sets		
	No Sharing	Sharing	Pct Saved
allroots	328	87	73.48%
052.alvinn	464	89	80.82%
01.qbsort	362	111	69.34%
15.trie	360	125	65.28%
04.bisect	352	78	77.84%
17.bintr	350	124	64.57%
anagram	696	159	77.16%
lex315	1088	260	76.10%
ks	1054	317	69.92%
05.eks	694	170	75.50%
08.main	1166	399	65.78%
09.vor	1842	633	65.64%
loader	1434	382	73.36%
129.compress	1152	195	83.07%
ft	1140	380	66.67%
football	5536	962	82.62%
compiler	3486	699	79.95%
assembler	3184	911	71.39%
yacr2	3960	782	80.25%
simulator	5180	1203	76.78%
099.go	—	4966	—
Average			73.78%

and 20.8% respectively. (`09.vor` had high averages in these categories and has the lowest effectiveness for this technique while `yacr2` had lower than average percentages and resulted in a higher speed-up.)

Over all benchmarks, this technique results in a significant speed-up, 2.80 on average. For our largest benchmark, `099.go`, the analysis did not run (due to insufficient memory) until we applied this optimization.

4.2 Worklists

The initial implementation of the analyses used an iterative algorithm for simplicity. After correctness was verified, a worklist-based implementation was used to improve efficiency. Two types of worklists were used: SEG node worklists and function worklists. Each function has a worklist of SEG nodes. The PCG has two worklists of functions: “current” and “next.” (The motivation for using two worklists will be described in Section 4.3.)

We use nested “while not empty” loops with the worklists — the outer loop visiting functions and the inner loop visiting SEG nodes. The worklist of functions initially contains all functions. On the first visit to a function, we initialize the function’s SEG node worklist with all SEG nodes in that function. If a SEG node’s *Out* set changes, all its SEG successors are placed on its function’s SEG node worklist. If a function’s entry set changes, it is placed on the “next” function worklist. If the exit set of a function changes, all calling functions are placed on the “next” function worklist and the calling call node(s) are placed on their respective function’s SEG node worklist. The analysis runs until all worklists are empty.

Table 3. Flow-sensitive analysis run time in seconds. Numbers in parentheses are the speed-up from the previous version to the left.

Benchmark Name	No Sharing		← Sharing →			Overall Speed-up
	← Iterating →		← Worklists →			
			Unsorted	Sorted		
	← No Forward Bind Filter →		Filter			
allroots (227 LOC)	1.41	0.54 (2.61)	0.21 (2.57)	0.24 (0.88)	0.10 (2.40)	14.10
052.alvinn (272 LOC)	1.03	0.39 (2.64)	0.09 (4.33)	0.10 (0.90)	0.10 (1.00)	10.30
01.qbsort (325 LOC)	8.22	4.16 (1.98)	1.81 (2.30)	1.19 (1.52)	0.75 (1.59)	10.96
15.trie (358 LOC)	4.59	2.26 (2.03)	0.83 (2.72)	0.72 (1.15)	0.45 (1.60)	10.20
04.bisect (463 LOC)	1.47	0.45 (3.27)	0.16 (2.81)	0.13 (1.23)	0.14 (0.93)	10.50
17.bintr (495 LOC)	2.20	1.19 (1.85)	0.33 (3.61)	0.32 (1.03)	0.32 (1.00)	6.88
anagram (650 LOC)	6.67	2.16 (3.09)	0.59 (3.66)	0.52 (1.13)	0.48 (1.08)	13.90
lex315 (733 LOC)	5.03	2.09 (2.41)	0.73 (2.86)	0.70 (1.04)	0.41 (1.71)	12.27
ks (782 LOC)	20.48	9.40 (2.18)	3.32 (2.83)	2.55 (1.30)	1.49 (1.71)	13.74
05.eks (1202 LOC)	9.21	2.87 (3.21)	0.92 (3.12)	0.87 (1.06)	0.67 (1.30)	13.75
08.main (1206 LOC)	96.37	44.80 (2.15)	18.30 (2.45)	12.39 (1.48)	3.16 (3.92)	30.50
09.vor (1406 LOC)	217.19	113.71 (1.91)	38.70 (2.94)	32.96 (1.17)	11.92 (2.77)	18.22
loader (1539 LOC)	176.49	58.37 (3.02)	26.71 (2.19)	21.20 (1.26)	3.81 (5.56)	46.32
129.compress (1934 LOC)	6.00	1.82 (3.30)	0.56 (3.25)	0.53 (1.06)	0.41 (1.29)	14.63
ft (2156 LOC)	80.94	37.07 (2.18)	14.04 (2.64)	11.25 (1.25)	5.09 (2.21)	15.90
football (2354 LOC)	275.63	61.30 (4.50)	26.87 (2.28)	22.75 (1.18)	2.70 (8.43)	102.09
compiler (2360 LOC)	39.31	12.87 (3.05)	4.19 (3.07)	4.11 (1.02)	3.70 (1.11)	10.62
assembler (3446 LOC)	668.25	240.46 (2.78)	119.31 (2.02)	97.26 (1.23)	9.69 (10.04)	68.96
yacr2 (3979 LOC)	377.11	86.50 (4.36)	26.76 (3.23)	25.93 (1.03)	10.64 (2.44)	35.44
simulator (4639 LOC)	511.49	146.35 (3.49)	82.41 (1.78)	61.84 (1.33)	10.12 (6.11)	50.54
099.go (29637 LOC)	—	228.18 —	74.01 (3.08)	65.23 (1.13)	9.83 (6.64)	23.21
Averages		2.80	2.84	1.16	3.09	25.38

Column 4 of Table 3 shows the improvement of the worklist-based implementation over the iterating version, both of which use shared alias sets. The result was an average speed-up of 2.84 over the iterating version, which produced an average speed-up of 2.78 over the nonshared iterating version.

4.3 Sorted Worklists

Using an iterating analysis of a forward data-flow problem, it is natural to process the nodes in topological order. The next enhancement was to use priority queues (based on a topological order¹⁶) for the SEG and function worklists.

Consider the case of a loop body that generates aliasing information. It would be best to process the loop body before moving on to the loop exit. Topological order alone does not give this property — we may process the loop exit before the loop body. (This would occur in Fig. 7 using the given node numbers as the topological order.) However, during the construction of the CFG, nodes for loop bodies are created before those nodes after the loop body. (Thus, node #7 would be created before node #3.) Since nodes are assigned numbers as they are created (which is performed in a topological order, except in the presence of gotos), using the nodes creation number as a priority ensures that loop bodies are processed first.

A result of using a single priority-based worklist of functions was that calling functions were visited before called functions. However, unlike SEG nodes, aliases can be propagated in both directions along a PCG edge. Thus, an optimal order of function visits is not apparent. In our benchmark suite, using a single priority-based worklist for functions provided only a marginal improvement over the iterating version.

To increase efficiency, we use two function worklists — “current” and “next.” While visiting functions on the “current” worklist, we place functions on the “next” worklist. This has the effect that once a set of functions is on the worklist, the visiting order is fixed in a topological order. When the “current” worklist is empty, we swap the “current” and “next” worklists and continue the analysis.

The fifth column of Table 3 reports the analysis time using sorted worklists for both SEG nodes and functions and the previous enhancements. This enhancement resulted in an average speed-up of 1.16 over the previous version, which used nonpriority-based worklists and shared alias sets.

4.4 ForwardBind Filtering

`ForwardBind()` is a function in our analysis that propagates alias relations from call nodes to the entry set of called functions. If needed, it creates alias relations for the formals from the actuals in the called function’s entry set and then unions in the call node’s *In* set with the called function’s entry set. Consider those alias relations that cannot be changed or used in the called function (for example, the

¹⁶ Topological order on a cyclic graph can be obtained by performing a depth-first search of the graph and then removing edges classified as back edges [24].

relation $\langle *l, a \rangle$ in Fig. 7, but are still propagated through the called function until they reach the exit set of the function. These relations are then propagated back to the call node’s *Out* set. In effect, the called function acts as an identity transfer function for those relations that are not relevant in the called function. Although correct, this is inefficient.

Our enhancement is not to propagate from the call node those alias relations that cannot be reached in a called function. To compute the set of alias relations that are not reachable in the called function, F , we first call **ForwardBind()**, which propagates alias relations into the entry set of F , $Entry_F$, as described above. We then view all alias relations in $Entry_F$ as a directed graph and remove from this graph all vertices (distinguished objects), and associated edges (alias relations), that cannot be reached from a global or a formal of F . These removed edges (alias relations) are simply unioned into the call node’s *Out* set directly. This can help limit the propagation effects of the unrealizable path problem.

The last column of Table 3 reports the effectiveness of this optimization. It provided the most dramatic speed-up, an average of 3.09 over the previous implementation, which used all other enhancements. Some programs, in particular **football** and **assembler**, had a much higher than average speed-up resulting from this filtering. These two programs both shared some common characteristics: they have a single function that has both an unusually high amount of pointer affecting statements and a very high number of called functions.

Figure 8 shows the effects of these optimizations in a dramatic way for the **loader** benchmark¹⁷. We collected the data presented in this graph by repeatedly executing the “**ps v**” command (under AIX 4.1.5) while each of the five FS analyses was running. We recorded the **SIZE** column, which gives the virtual size of the data section of the process; this will capture all heap allocated memory usage during the analysis. The x-axis of the chart is simply marked off in samples (a sample is a single call to “**ps v**”).

As the majority of our heap allocations are used to represent alias relations, the resulting memory usage can be interpreted as the number of alias relations stored by the analysis while running. The “No Sharing” version shows a characteristic curve; it grows quickly early on, but then levels off as the analysis reaches its fixed point. The difference between the No Sharing and Sharing versions shows how the large reduction in the number of alias sets reduced the number of alias relations that were stored.

The cumulative effect of all five optimizations was an average speed-up of 25.38. This illustrates the benefits that can be obtained by limiting the propagation of extraneous alias relations and the number of visits to functions and nodes.

5 Other Related Work

This section describes other related work not mentioned in Section 3.3.

¹⁷ The other benchmarks had similarly shaped graphs.

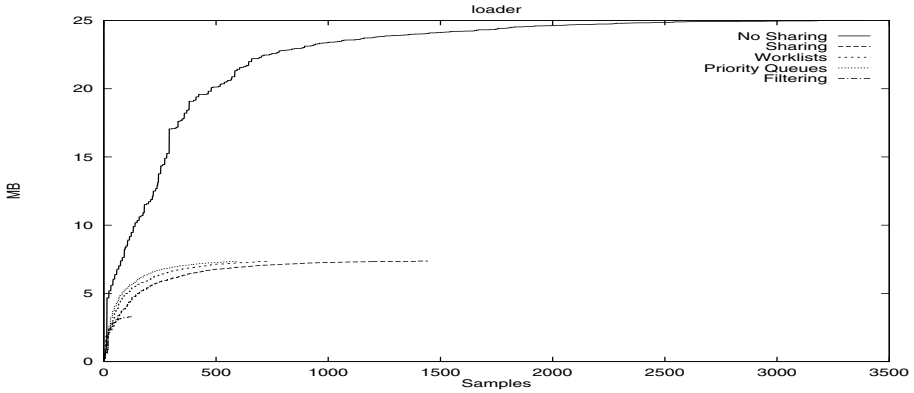


Fig. 8. Memory usage over time for the `loader` benchmark.

Ruf [36] presents an empirical study of two algorithms: a flow-sensitive algorithm similar to the one we have implemented, and a context-sensitive version of the same algorithm. His results showed that the context-sensitive algorithm did not improve precision for pointers where they are dereferenced, but cautioned that this may be a characteristic of the benchmark suite analyzed. Wilson and Lam [48, 47] present an algorithm for performing context-sensitive analysis that avoids redundant analyses of functions for similar calling contexts. Ghiya and Hendren [17] present empirical data showing how points-to and connection analyses can improve traditional transformations, array dependence testing, and program understanding.

Ruf [37] describes a program partitioning technique that is used for a flow-sensitive points-to analysis, achieving a storage savings of 1.3–7.2 over existing methods. Diwan et al. [13] provide static and dynamic measurements of the effectiveness of three flow-insensitive analyses for a type-safe language (Modula-3). With the exception of AT, all three algorithms are less precise than the versions we have studied. Zhang et al. [50] report the effectiveness of applying different pointer aliasing algorithms to different parts of a program.

Hasti and Horwitz [18] present a pessimistic algorithm that attempts to increase the precision of a flow-insensitive analysis by iterating over a flow-insensitive analysis and an SSA [10] construction. No empirical results are reported. Horwitz [23] defines precise flow-insensitive alias analysis and proves that, even in the absence of dynamic memory allocation, computing it is NP-hard.

6 Conclusions

This work has described an empirical study of four pointer alias analysis algorithms that use varying degrees of flow-sensitivity. We have found that

- the address-taken analysis, although efficient, is unlikely to provide sufficient precision;
- the flow-insensitive analysis with kill is not beneficial;
- the precision of the flow-insensitive analysis is identical to that of the flow-sensitive analysis in 12 of 21 programs in our benchmark suite;
- most published implementations of flow-sensitive pointer analysis have equivalent precision.

Although the flow-sensitive analysis efficiently analyzed a program on the order of 30K LOCs, further benchmarks are needed to see if this property generalizes. We have also empirically demonstrated how various implementation strategies result in significant analysis-time speed-up.

7 Acknowledgements

We thank Michael Burke, Michael Karasick, and Lee Nackman for their support of this work. We also thank Todd Austin, Bill Landi, and Rakesh Ghiya for making their benchmarks available.

Bill Landi, Laurie Hendren, Erik Ruf, Barbara Ryder, and Bob Wilson provided useful details concerning their implementations. Michael Burke, Paul Carini, and Jong-Deok Choi provided useful discussions regarding the algorithms described in [8]. Discussions with Manuel Fähndrich led to reporting intermediate read dereferences, which were not considered in [20].

Harini Srinivasan designed and implemented the initial control flow graph builder, an important early system component. NPIC group members Robert Culley, Lynne Delesky, Lap Chung Lam, Giampaolo Lauria, Mark Nicosia, Joseph Perillo, Keith Sanders, Truong Vu, and Ming Wu assisted with the implementation and testing of the system. David Bacon helped with the initial design of the program call graph representation.

Michael Burke, Jong-Deok Choi, John Field, G. Ramalingam, Harini Srinivasan, Lauren Treacy, and the anonymous referees provided useful comments on earlier drafts of this paper.

References

- [1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. Available at ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z. [70, 70]
- [2] Todd Austin. Pointer-intensive benchmark suite, version 1.1. <http://www.cs.wisc.edu/~austin/ptr-dist.html>, 1995. [62]
- [3] Subra Balan and Walter Bays. Spec announces new benchmark suites cint92 and cfp92. Technical report, Systems Performance Evaluation Cooperative, March 1992. *SPEC Newsletter* 4(1). [62]
- [4] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer*

- Science*, 892, pages 234–250. Springer-Verlag, 1995. Proceedings from the 7th Workshop on Languages and Compilers for Parallel Computing. Extended version published as Research Report RC 19546, IBM T. J. Watson Research Center, September 1994. [58](#), [58](#), [61](#), [62](#)
- [5] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Interprocedural pointer alias analysis. Research Report RC 21055, IBM T. J. Watson Research Center, December 1997. [58](#), [58](#), [58](#), [59](#), [59](#), [60](#), [60](#), [61](#), [61](#), [62](#)
 - [6] Paul Carini, Michael Hind, and Harini Srinivasan. Flow-sensitive interprocedural type analysis for C++. Research Report RC 20267, IBM T. J. Watson Research Center, November 1995. [59](#)
 - [7] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990. *SIGPLAN Notices* 25(6). [60](#), [60](#), [65](#)
 - [8] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 232–245, January 1993. [58](#), [60](#), [60](#), [78](#)
 - [9] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *18th Annual ACM Symposium on the Principles of Programming Languages*, pages 55–66, January 1991. [59](#), [72](#)
 - [10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, pages 451–490, October 1991. [77](#)
 - [11] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994. *SIGPLAN Notices*, 29(6). [60](#)
 - [12] Amer Diwan. Personal communication, August 1997. [69](#)
 - [13] Amer Diwan, Kathryn S. McKinley, and J. Elliot B. Moss. Type-based alias analysis. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 106–117, June 1998. *SIGPLAN Notices*, 33(5). [77](#)
 - [14] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994. *SIGPLAN Notices*, 29(6). [60](#), [60](#), [62](#), [70](#), [71](#)
 - [15] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Lecture Notes in Computer Science*, 1033, pages 515–533, August 1995. Proceedings from the 8th Workshop on Languages and Compilers for Parallel Computing. [60](#)
 - [16] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag or a cyclic graph? A shape analysis for heap-directed pointers in C. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 1–15, January 1996. [60](#)
 - [17] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *25th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 121–133, January 1998. [77](#)
 - [18] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 97–105, June 1998. *SIGPLAN Notices*, 33(5). [77](#)

- [19] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990. [60](#)
- [20] Michael Hind and Anthony Pioli. An empirical comparison of interprocedural pointer alias analyses. Research Report RC 21058, IBM T. J. Watson Research Center, December 1997. [78](#)
- [21] Michael Hind and Anthony Pioli. Assessing the effects of the flow-sensitivity on pointer alias analyses (extended version). Technical Report 98-104, SUNY at New Paltz, June 1998. [61](#) [65](#) [65](#) [67](#)
- [22] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989. *SIGPLAN Notices* 24(6). [60](#) [60](#)
- [23] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-Hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997. [77](#)
- [24] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976. [75](#)
- [25] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992. [57](#)
- [26] William Landi. Personal communication, October 1997. [62](#) [69](#) [69](#)
- [27] William Landi and Barbara Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992. *SIGPLAN Notices* 27(6). [60](#) [69](#) [70](#)
- [28] William Landi, Barbara Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993. *SIGPLAN Notices* 28(6). [62](#) [62](#) [69](#)
- [29] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, 1988. *SIGPLAN Notices*, 23(7). [60](#)
- [30] Thomas Marlowe, William Landi, Barbara Ryder, Jong-Deok Choi, Michael Burke, and Paul Carini. Pointer-induced aliasing: A clarification. *SIGPLAN Notices*, 28(9):67–70, September 1993. [60](#)
- [31] Thomas J. Marlowe, Barbara G. Ryder, and Michael G. Burke. Defining flow sensitivity in data flow problems. Technical Report RC 20138, IBM T. J. Watson Research Center, July 1995. [58](#)
- [32] Lee R. Nackman. Codestore and incremental C++. *Dr. Dobbs Journal*, pages 92–95, December 1997. [61](#)
- [33] Anthony Pioli. Conditional pointer aliasing and constant propagation. Master's thesis, SUNY at New Paltz, 1998. In preparation. [61](#)
- [34] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(6):1467–1471, November 1994. [57](#)
- [35] G. Ramalingam. On sparse evaluation representations. In Pascal Van Hentenryck, editor, *Lecture Notes in Computer Science*, 1302, pages 1–15. Springer-Verlag, 1997. Proceedings from the 4th International Static Analysis Symposium. [72](#)
- [36] Erik Ruf. Context-insensitive alias analysis reconsidered. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995. *SIGPLAN Notices*, 30(6). [62](#) [62](#) [62](#) [65](#) [70](#) [70](#) [70](#) [71](#) [77](#)

- [37] Erik Ruf. Partitioning dataflow analyses using types. In *24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 15–26, January 1997. [77](#)
- [38] Erik Ruf. Personal communication, October 1997. [62](#), [62](#), [70](#)
- [39] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 16–31, January 1996. [60](#)
- [40] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998. [60](#)
- [41] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In Pascal Van Hentenryck, editor, *Lecture Notes in Computer Science*, 1302, pages 16–34. Springer-Verlag, 1997. Proceedings from the *4th International Static Analysis Symposium*. [69](#), [70](#)
- [42] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive point-to analysis. In *24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 1–14, January 1997. [61](#), [70](#), [70](#)
- [43] D. Soroker, M. Karasick, J. Barton, and D. Streeter. Extension mechanisms in Montana. In *8th IEEE Israeli Conference on Software and Systems*, pages 119–128, June 1997. [61](#)
- [44] SPEC. SPEC CPU95, Version 1.0. Standard Performance Evaluation Corporation, <http://www.specbench/org>, August 1995. [62](#)
- [45] Bjarne Steensgaard. Points-to analysis in almost linear time. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 32–41, January 1996. [61](#), [70](#), [70](#), [70](#)
- [46] Philip A. Stocks, Barbara G. Ryder, William A. Landi, and Sean Zhang. Comparing flow and context sensitivity on the modifications-side-effects problem. In *International Symposium on Software Testing and Analysis*, pages 21–31, March 1998. [69](#), [70](#), [70](#)
- [47] Robert P. Wilson. *Efficient Context-Sensitive Pointer Analysis for C Programs*. PhD thesis, Stanford University, December 1997. [77](#)
- [48] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995. *SIGPLAN Notices*, 30(6). [60](#), [77](#)
- [49] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *4th Symposium on the Foundations of Software Engineering*, pages 81–92, October 1996. [70](#), [70](#)
- [50] Sean Zhang, Barbara G. Ryder, and William Landi. Experiments with combined analysis for pointer aliasing. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 11–18, June 1998. [70](#), [77](#)

Analysis of Normal Logic Programs

François Fages and Roberta Gori

LIENS CNRS, Ecole Normale Supérieure,
45 rue d'Ulm, 75005 Paris, France,
`fages@dmi.ens.fr`

Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56125 Pisa, Italy
`gori@di.unipi.it`
Ph.: +39-50-887248 Fax: +39-50-887226

Abstract. In this paper we present a dataflow analysis method for normal logic programs interpreted with negation as failure or constructive negation. We apply our method to a well known analysis for logic programs: the depth(k) analysis for approximating the set of computed answers. The analysis is correct w.r.t. SLDNF resolution and optimal w.r.t. constructive negation.

Keywords: Abstract interpretation, static analysis, logic programming, constructive negation.

1 Introduction

Important results have been achieved for static analysis using the theory of abstract interpretation [6]. Abstract interpretation is a general theory for specifying and validating program analysis.

A key point in abstract interpretation is the choice of a reference semantics from which one can abstract the properties of interest. While it is always possible to use the operational semantics, it is possible to get rid of useless details, by choosing a more abstract semantics as reference semantics. In the case of definite logic programs, much work has been done in this sense. Choosing the most abstract logical least model semantics limits the analysis to type inference properties, that approximate the ground success set. Non-ground model semantics have thus been developed, under the name of S-semantics [2], and proved useful for a wide variety of goal-independent analysis ranging from groundness, to sharing, call patterns, etc. All the intermediate fixpoint semantics between the most abstract logical one and the most concrete operational one, form in fact a hierarchy related by abstract interpretation, in which one can define a notion of the best reference semantics [12] for a given analysis.

On the other hand, less work has been done on the analysis of normal logic programs, although the finite failure principle, and hence SLDNF resolution, are standard practice. The most significant paper on the analysis of normal

logic programs, using the theory of abstract interpretation, is the one by Marriott and Søndergaard [19], which proposes a framework based on Fitting's least three-valued model semantics [11]. Since this reference semantics is a ground semantics, the main application of this framework is type analysis. Marriott and Søndergaard already pointed out that a choice of a different reference semantics could lead to an improved analysis. Fitting's least three-valued model semantics is, in fact, an abstraction (a non recursively enumerable one, yet easier to define) of Kunen's three-valued logical semantics [14] which is more faithful to SLDNF resolution [15] and complete w.r.t. constructive negation.

These are exactly the directions along which we try to improve the results of Marriott and Søndergaard. We consider the inference rule of constructive negation, which provides normal logic programs with a sound and complete [21] operational semantics w.r.t. Kunen's logical semantics [14]. We propose an analysis method for normal logic programs interpreted with constructive negation, based on the generalized S-semantics given in [9] and on the hierarchy described in [10]. We present here an analysis based on the $\text{depth}(k)$ domain which approximates the computed answers obtained by constructive negation and therefore the three-valued consequences of the program completion and CET (Clark's equational theory). Other well known analyses for logic programs can be extended to normal logic programs. For example, starting from a suitable version of Clark's semantics a groundness analysis was defined which is correct and optimal w.r.t. constructive negation. Here, for lack of space, we present only the $\text{depth}(k)$ analysis. We show that it is correct and also optimal w.r.t. constructive negation. Finally we give an example which shows that in the case of type inference properties our semantics yields a result which is more precise than the one obtained by Marriott and Søndergaard.

From the technical point of view, the contribution of the paper is the definition of a normal form for first order constraints on the Herbrand Universe, which is suitable for analysis. In fact the normal form allows us to define an abstraction function which is a congruence w.r.t. the equivalence on constraints induced by the Clark's equality theory.

The paper is organized as follows. In Sect. 2 we introduce some preliminary notions on constructive negation. In Sect. 3 we define a *normal* form on the concrete domain of constraints in order to deal, with equivalence classes of constraints w.r.t. the Clark's equational theory. Section 4 defines the abstract domain and abstract operator and show its correctness and optimality (under suitable assumptions on the depth of the cut) w.r.t. the concrete one. Finally, Subsect. 4.5 shows an example.

2 Preliminaries

The reader is assumed to be familiar with the terminology of and the basic results in the semantics of logic programs [11, 17] and with the theory of abstract interpretation as presented in [6, 7].

2.1 Normal Logic Programs and Constructive Negation

We consider the equational version of normal logic programs, where a *normal program* is a finite set of clauses of the form $A \leftarrow c | L_1, \dots, L_n$, where $n \geq 0$, A is an atom, called the head, c is a conjunction of equalities, and L_1, \dots, L_n are literals. The *local variables* of a program clause are the free variables in the clause which do not occur in the head. With $Var(A)$ we intend the free variables in the atom A .

In order to deal with constructive negation, we need to consider the domain \mathcal{C} of full first-order equality constraints on the structure \mathcal{H} of the Herbrand domain. Assuming an infinite number of function symbols in the alphabet, *Clark's equational theory* (CET) provides a complete decidable theory for the constraint language [18,14], i.e.

1. (soundness) $\mathcal{H} \models CET$,
2. (completeness) for any constraint c , either $CET \models \exists c$ or $CET \models \neg \exists c$.

A constraint is in *prenex form* if all its quantifiers are in the head. The set of free variables in a constraint c is denoted by $Var(c)$. For a constraint c , we shall use the notation $\exists c$ (resp. $\forall c$) to represent the constraint $\exists X c$ (resp. $\forall X c$) where $X = Var(c)$.

A *constrained atom* is a pair $c|A$ where c is an \mathcal{H} -solvable constraint such that $Var(c) \subseteq Var(A)$. The set of constrained atoms is denoted by B . A *constrained interpretation* is a subset of B . A *three-valued* or *partial constrained interpretation* is a pair of constrained interpretations $\langle I^+, I^- \rangle$, representing the constrained atoms which are true and false respectively (note that because of our interest in abstract interpretations we do not impose any consistency condition between I^+ and I^-).

Constructive negation is a rule of inference introduced by Chan for normal logic programs in [3], which provides normal logic programs with a sound and complete [21] operational semantics w.r.t. Kunen's logical semantics [14]. In logic programming, Kunen's semantics is simply the set of three-valued consequences of the program's completion and the theory *CET*.

The S -semantics of definite logic programs [2] has been generalized to normal logic programs in [9] for a version of constructive negation, called constructive negation by pruning. The idea of the fixpoint operator, which captures the set of computed answer constraints, is to consider a *non-ground finitary* (hence continuous) version of Fitting's operator. Here we give a definition of the operator $T_P^{\mathcal{B}_{\mathcal{D}}}$ which is parametric w.r.t. the domain $\mathcal{B}_{\mathcal{D}}$ of constrained atoms and the operations on constraints on the domain \mathcal{D} .

Definition 1. Let P be a normal logic program. $T_P^{\mathcal{B}_{\mathcal{D}}}$ is an operator over $\mathcal{P}(\mathcal{B}_{\mathcal{D}}) \times \mathcal{P}(\mathcal{B}_{\mathcal{D}})$ defined by

$$\begin{aligned}
 T_P^{\mathcal{B}_D}(I)^+ &= \{c|p(X) \in \mathcal{B}_D : \text{there exists a clause in } P \text{ with local variables } Y, \\
 &\quad C = p(X) \leftarrow d|A_1, \dots, A_m, \neg A_{m+1}, \dots, \neg A_n. \\
 &\quad c_1|A_1, \dots, c_m|A_m \in I^+, c_{m+1}|A_{m+1}, \dots, c_n|A_n \in I^- \\
 &\quad \text{such that } c = \exists Y (d \ \overline{\wedge} c_1 \overline{\wedge} \dots \overline{\wedge} c_n)\} \\
 T_P^{\mathcal{B}_D}(I)^- &= \{c|p(X) \in \mathcal{B}_D : \text{for each clause defining } p \text{ in } P \text{ with loc. var. } Y_k, \\
 &\quad C_k = p(X) \leftarrow d_k|A_{k,1}, \dots, A_{k,m_k}, \beta_k. \\
 &\quad \text{there exist } e_{k,1}|A_{k,1}, \dots, e_{k,m_k}|A_{k,m_k} \in I^-, \\
 &\quad n_k \geq m_k, e_{k,m_k+1}|A_{k,m_k+1}, \dots, e_{k,n_k}|A_{k,n_k} \in I^+, \\
 &\quad \text{where for } m_{k+1} \leq j \leq n_k, \neg A_{k,j} \text{ occurs in } \beta_k, \\
 &\quad \text{such that } c = \overline{\wedge}_k \overline{\vee} Y_k (\neg d_k \overline{\vee} e_{k,1} \dots \overline{\vee} e_{k,n_k})\}.
 \end{aligned}$$

where the operations $\exists, \overline{\vee}, \neg, \overline{\vee}, \overline{\wedge}$, are the corresponding operations on the constraint domain of \mathcal{D} .

In the case of a normal logic program, the operator T_P^B defines a generalized S-semantics which is fully abstract w.r.t. the computed answer constraints with constructive negation by pruning [9]. By soundness it approximates also the set of computed answer constraints under the SLDNF resolution rule, or under the Prolog strategy.

In [10] we have shown that this operator defines a hierarchy of reference semantics related by abstract interpretation, that extends the hierarchy defined by Giacobazzi for definite logic programs [12]. Here we show the use of the hierarchy for the static analysis of normal logic programs.

3 Normal Forms in CET

Unlike the semantics in Marriott and Søndergaard's framework, our reference semantics is a non ground semantics and has to deal with first-order equality constraints. The first problem that arises is to define a normal form for such constraints on the Herbrand domain, so that abstraction functions on constrained atoms can be defined. In general, in fact, given a theory th , we are interested in working with equivalence classes of constraints w.r.t. the equivalence of the constraints in th . Namely c is equivalent to c' if $th \models c \leftrightarrow c'$. Therefore we need the abstraction function on the concrete constraint domain to be a congruence. This is a necessary property since it permits to be independent from the syntactic form of the constraints.

Dealing with normal logic programs, we need to achieve this property in CET. We thus need to introduce a normal form for first-order equality constraints, in a similar way to what has been done for the analysis of definite programs where the normal form is the unification solved form [16]. Here we shall define a new notion of “false-simplified” normal forms, making use of Colmerauer's solved forms for inequalities [4], Maher's transformations for first-order constraints [18] and an extended disjunctive normal form [13].

First let us motivate the need of a “false-simplified” form. Let us call a *basic constraint* an equality or an inequality between a variable and a term. The abstraction function will be defined inductively on the basic constraints, and it

will sometimes (e.g. for groundness analysis) abstract to *true* some inequalities. Consider, for example, the following constraint $d = \forall X(Y = b \wedge X \neq f(a))$. d is \mathcal{H} -equivalent to *false*. If the abstraction of $X \neq f(a)$ is *true* then the abstraction of d will be the abstraction of $Y = b$, which cannot be \mathcal{H} -equivalent to the abstraction of *false*. Therefore we need to define a normal form where the constraints which are \mathcal{H} -equivalent to *false*, are eliminated.

Definition 2. Consider a constraint d in prenex disjunctive form, $d = \Delta(\vee_i A_i)$, where Δ is a sequence of quantified variables and $\vee_i A_i$ is a finite disjunction. d is in a *false-simplified* form if, either there does not exist a proper subset I of the i 's such that $\mathcal{H} \models \Delta(\vee_i A_i) \leftrightarrow \Delta(\vee_{i \in I} A_i)$, or such an I exists and there exists also a subset K of I , such that $\vee_{j \notin I} A_j$ is \mathcal{H} -equivalent to $\vee_{k \in K} A_k$.

The latter condition assures that we really eliminate constraints that are \mathcal{H} -equivalent to *false* and that are not just redundant in the constraint. Now the existence of a *false-simplified* form for any constraint can be proved simply with the following:

Algorithm 3 Input: a constraint in prenex disjunctive form $d = \Delta(\vee_i A_i)$.

Let us call U the set of the indices i 's in $d = \Delta(\vee_i A_i)$.

1. Let I and J be the partition of U such that $i \in I$ if $\mathcal{H} \models \exists \Delta(A_i)$, otherwise $i \in J$.
2. Repeat $I := I \cup S$ as long as there exists an $S \subseteq J$ such that $\mathcal{H} \models \exists \Delta(\vee_{i \in S} A_i)$ and for all $j \in S$ $\mathcal{H} \not\models \exists \Delta(\vee_{i \in (S \setminus \{j\})} A_i)$.
3. Let $S \subseteq J \setminus I$ be any minimal set such that $\mathcal{H} \models \exists \Delta(\vee_{s \in S} A_s \vee_{i \in I} A_i)$ and $\mathcal{H} \models \Delta(\vee_{s \in S} A_s \vee_{i \in I} A_i) \leftrightarrow d$, do $I := I \cup S$.
4. Output: $\Delta(\vee_{i \in I} A_i)$.

The idea of the algorithm is to find a subset of the conjunctions A_i 's (those with $i \in I$) such that $\Delta(\vee_{i \in I} A_i)$ is in *false-simplified* form and it is \mathcal{H} -equivalent to $\Delta(\vee_i A_i)$. In the first step we select the A_i 's such that $\Delta(A_i)$ is \mathcal{H} -satisfiable. In this case, in fact, A_i cannot be \mathcal{H} -equivalent to *false* and it can be put in the set I . In the second step from the remaining A_i 's we select the set of A_i 's such that their Δ quantified disjunction is \mathcal{H} -satisfiable, since we check that all the A_i 's are necessary for the quantified disjunction to be \mathcal{H} -satisfiable, the considered A_i 's can not be \mathcal{H} -equivalent to *false*. At the end of this process, if the resulting constraint is \mathcal{H} -equivalent to the input constraint, we stop. Otherwise, we add a minimum number of the not yet selected A_i 's such the $\Delta(\vee_i A_i)$ for the selected i 's is \mathcal{H} -equivalent to the input constraint. Since we add a minimum number of not yet selected A_i 's, we are sure that the resulting constraint is in *false-simplified* form. Example 4 shows how the algorithm 3 works on two examples.

Example 4. 1. Input: $c_1 = \forall T(A_1 \vee A_2 \vee A_3 \vee A_4)$, $A_1 = (T = f(H) \wedge Y = a)$, $A_2 = (T \neq f(a) \wedge Y = b)$, $A_3 = (Y \neq g(H, T))$, $A_4 = (T \neq a \wedge Y = a)$.

$I_1 = \{3\}$.

$I_2 = \{3, 1, 2\}$.

$I_3 = \{3, 1, 2\}$ (since $\mathcal{H} \models \forall T(\vee_{i \in I_3} A_i) \leftrightarrow c_1$).

Output: $\forall T(A_1 \vee A_2 \vee A_3)$.

2. Input: $c_2 = \forall T(A'_1 \vee A'_2 \vee A'_3)$, $A'_1 = (T \neq f(U) \wedge T \neq f(V))$,
 $A'_2 = (T = H)$, $A'_3 = (U \neq V \wedge T = f(a))$.
 $I_1 = \{\}$.
 $I_2 = \{1, 2\}$.
 $I_3 = \{1, 2, 3\}$ (since $\mathcal{H} \not\models \forall T(\bigvee_{i \in I_2} A'_i) \leftrightarrow c_2$).¹
 Output: $\forall T(A'_1 \vee A'_2 \vee A'_3)$.

Theorem 5. *For any input constraint $c = \Delta(\bigvee_i A_i)$, algorithm 3 terminates and computes a false-simplified form logically equivalent to c .*

Note that all the false-simplified forms of a constraint c are \mathcal{H} -equivalent.

Now the intuitive idea for a normal form is the following. We put a constraint in prenex form and we compute the disjunctive form of its quantifier free part. We make equality and inequality constraints interact in every conjunction of the disjunctive form and then we compute the false-simplified form for the resulting constraint. The problem is that if we consider a standard disjunctive normal form, we would not be able to see explicitly all the relations between constraints in disjunctions. Consider, for example, the constraint $(X = f(H) \vee (H \neq f(a)))$. This constraint is equivalent, therefore \mathcal{H} -equivalent, to the constraint $((X = f(f(a)) \wedge H = f(a)) \vee H \neq f(a))$. Note that the equality $H = f(a)$ is not explicit in the first disjunction. Since the abstraction function will act on the terms of the disjunction independently, this could cause a problem. This is why we will use a well known *extended disjunctive form* defined for Boolean algebra and applied, in our case, to the algebra of quantifier free constraints.

In the next theorem with B_i we denote basic equality or inequality constraints ($X = t$ or $X \neq t$). For any B_i let $B_i^{false} = \neg B_i$ and $B_i^{true} = B_i$.

Theorem 6. [13] *For every Boolean formula ϕ on basic equality or inequality constraints B_1, \dots, B_n ,
 $\phi \leftrightarrow \psi$ where $\psi = (\bigvee_{(a_1, \dots, a_n) \in \{false, true\}^n} \phi(a_1, \dots, a_n) \wedge B_1^{a_1} \wedge \dots \wedge B_n^{a_n})$.*

Note that ψ is a formula in disjunctive form. ψ has in fact a particular disjunctive form where each conjunction contains all the basic constraints (possibly negated) of ϕ . This is why, this form is able to capture all the possible relations between the different terms of a disjunction.

We will call the formula ψ the *extended disjunctive normal form* (dnf) of ϕ . The next example shows how the extended disjunctive normal form works on a constraint c_1 .

Example 7. $c_1 = (X = f(H) \vee H \neq f(a))$.
 $dnf(c_1) = ((X = f(f(a)) \wedge H = f(a)) \vee$
 $(X = f(H) \wedge H \neq f(a)) \vee (X \neq f(H) \wedge H \neq f(a)))$.

Note that although c_1 , $dnf(c_1)$ and $((X = f(f(a)) \wedge H = f(a)) \vee H \neq f(a))$ are \mathcal{H} -equivalent, $dnf(c_1)$ is the most “complete” in the sense that it shows syntactically all the relations between constraints in disjunctions.

¹ $U = b, H = f(b), V = a$, in fact, is an assignment (for the free variables of c_2), which is a solution of c_2 but is not a solution of $\forall T(\bigvee_{i \in I_2} A'_i)$.

We can now define the normal form, $Res(c)$, of a first-order equality constraint c , as the result of the following steps:

1. Put the constraint c in prenex form obtaining $\Delta(c_1)$, where Δ is a sequence of quantified variables and c_1 is the quantifier free part of c .
2. Compute $dnf(c_1) = \vee(A_i)$,
3. Simplify each conjunction A_i obtaining $A'_i = ResConj(A_i)$,
4. Return a *false*-simplified form of the constraint $\Delta(\vee A'_i)$.

where the procedure for simplifying each conjunction is based on Maher's canonical form [18] and Colmerauer's simplification algorithm for inequalities [4]. The procedure performs the following steps,

$$ResConj(A)$$

1. compute a unification solved form for the equalities in the conjunction A
2. for each equality $X = t$ in A , substitute X by t at each occurrence of X in the inequalities of conjunction A .
3. simplify the inequalities by applying the following rules,
 - a) replace $f(t_1, \dots, t_n) \neq f(s_1, \dots, s_n)$ by $t_1 \neq s_1 \vee \dots \vee t_n \neq s_n$.
 - b) replace $f(t_1, \dots, t_n) \neq g(s_1, \dots, s_n)$ by *true*.
 - c) replace $t \neq x$ by $x \neq t$ if t is not a variable.
 obtaining A' ,
4. if A' is a conjunction then return A' .
5. otherwise compute $dnf(A) = \vee(A_i)$ and return $\vee ResConj(A_i)$.

It is worth noting that the previous algorithm terminates since each constraint contains a finite number of inequalities.

Example 8 shows how the procedure Res computes the normal form of some constraints.

Example 8.

$$\begin{aligned}
 1. \quad c &= (X = f(Y) \wedge (Y = a \vee Y = f(a)) \wedge \forall U \, X \neq f(f(U))). \\
 c_1 &= \forall U (X = f(Y) \wedge (Y = a \vee Y = f(a)) \wedge X \neq f(f(U))). \\
 c_2 &= \forall U ((X = f(Y) \wedge Y \neq a \wedge Y = f(a) \wedge X \neq f(f(U))) \vee \\
 &\quad (X = f(Y) \wedge Y = a \wedge Y \neq f(a) \wedge X \neq f(f(U))) \vee \\
 &\quad (X = f(Y) \wedge Y = a \wedge Y = f(a) \wedge X \neq f(f(U)))). \\
 c_{3.1} &= \forall U ((X = f(f(a)) \wedge Y \neq a \wedge Y = f(a) \wedge X \neq f(f(U))) \vee \\
 &\quad (X = f(a) \wedge Y = a \wedge Y \neq f(a) \wedge X \neq f(f(U)))). \\
 c_{3.2} &= \forall U ((X = f(f(a)) \wedge f(a) \neq a \wedge Y = f(a) \wedge f(f(a)) \neq f(f(U))) \vee \\
 &\quad (X = f(a) \wedge Y = a \wedge a \neq f(a) \wedge f(a) \neq f(f(U)))). \\
 c_{3.3} &= \forall U ((X = f(f(a)) \wedge Y = f(a) \wedge a \neq U) \vee \\
 &\quad (X = f(a) \wedge Y = a \wedge a \neq f(U))). \\
 c_4 &= (X = f(a) \wedge Y = a).
 \end{aligned}$$

$$\begin{aligned}
 2. \quad \bar{c} &= (X = f(Z, S) \wedge U = (f(H), H) \wedge S = a \wedge X \neq U). \\
 c_2 &= c_1 = \bar{c}. \\
 c_{3.1} &= (X = f(Z, a) \wedge U = f(f(H), H) \wedge S = a \wedge X \neq U). \\
 c_{3.2} &= (X = f(Z, a) \wedge U = f(f(H), H) \wedge S = a \wedge f(Z, a) \neq f(f(H), H)). \\
 c_{3.3} &= (X = f(Z, a) \wedge U = f(f(H), H) \wedge S = a \wedge (Z \neq f(H) \vee H \neq a). \\
 c_{3.4} &= A_1 \vee A_2 \vee A_3. \\
 A_1 &= (X = f(Z, a) \wedge U = f(f(H), H) \wedge S = a \wedge Z = f(H) \wedge H \neq a). \\
 A_2 &= (X = f(Z, a) \wedge U = f(f(H), H) \wedge S = a \wedge Z \neq f(H) \wedge H \neq a). \\
 A_3 &= (X = f(Z, a) \wedge U = f(f(H), H) \wedge S = a \wedge Z \neq f(H) \wedge H = a). \\
 \\
 ResConj(A_1) &= \bar{A}_1 \quad ResConj(A_2) = A_2 \quad ResConj(A_3) = \bar{A}_3. \\
 \bar{A}_1 &= (X = f(f(H), a) \wedge U = f(f(H), H) \wedge S = a \wedge Z = f(H) \wedge H \neq a). \\
 \bar{A}_3 &= (X = f(Z, a) \wedge U = f(f(a), a) \wedge S = a \wedge Z \neq f(a) \wedge H = a). \\
 c_4 &= \bar{A}_1 \vee A_2 \vee \bar{A}_3.
 \end{aligned}$$

Note that all the steps in *ResConj* and *Res* preserve the \mathcal{H} -equivalence, the third step of *ResConj* is Colmerauer's simplification algorithm for inequalities [4], the first and second transformations are the usual ones for CET formulas [18], while the second step of *Res* is the extended disjunctive normal transformation [13]. Hence we get:

Proposition 9. $\mathcal{H} \models \phi \leftrightarrow Res(\phi)$.

Our concrete constraints domain \mathcal{NC} will be the subset of constraints in \mathcal{C} which are in *normal* form. The concrete operations on \mathcal{NC} will be thus defined using the normal form:

Definition 10. Let $c_1, c_2 \in \mathcal{NC}$,

$$\begin{aligned}
 c_1 \wedge^c c_2 &= Res(c_1 \wedge c_2) & c_1 \vee^c c_2 &= Res(c_1 \vee c_2) \\
 \neg^c c_1 &= Res(\neg c_1) & \exists^c X \, c_1 &= \exists X \, c_1 & \forall^c X \, c_1 &= \forall X \, c_1
 \end{aligned}$$

We denote by \mathcal{B} the set of constrained atoms with constraints in \mathcal{NC} , and by (\mathcal{I}, \subseteq) the complete lattice of (not necessarily consistent) partial constrained interpretations formed over \mathcal{B} .

4 Depth(k) Analysis for Constructive Negation

The idea of depth(k) analysis was first introduced in [20]. The domain of depth(k) analysis was then used in order to approximate the ground success and failure sets for normal programs in [19].

We follow the formalization of [5] for positive logic programs. We want to approximate an infinite set of computed answer constraints by means of a constraint depth(k) cut, i.e. constraints where the equalities and inequalities are between variables and terms which have a depth not greater than k .

Our concrete domain is the complete lattice of partial constrained interpretations (\mathcal{I}, \subseteq) of the previous section. Since our aim is to approximate the computed answer constraints, the fixpoint semantics we choose in the hierarchy [10] is the one which generalizes the S -semantics to normal logic programs, the $T_P^{\mathcal{B}_P}$ operator (cf def. [1]). The version we consider here is the one defined on the domain \mathcal{B} with the concrete operations in \mathcal{NC} , \wedge^c , \vee^c , \neg^c , \exists^c , \forall^c , (the $T_P^{\mathcal{B}}$ operator).

4.1 The Abstract Domain

Terms are cut by replacing each-subterm rooted at depth greater than k by a new fresh variable taken from a set W , (disjoint from V). The $\text{depth}(k)$ terms represent each term obtained by instantiating the variables of W with terms built over V .

Consider the depth function $|| : \text{Term} \rightarrow \text{Term}$ such that

$$|t| = \begin{cases} 1 & \text{if } t \text{ is a constant or a variable} \\ \max\{|t_1|, \dots, |t_n|\} + 1 & \text{if } t = f(t_1, \dots, t_n) \end{cases}$$

and a given positive integer k . The abstract term $\alpha_k(t)$ is the term obtained from the concrete one by substituting a fresh variable (belonging to W) to each subterm t' in t , such that $|t| - |t'| = k$.

Consider now the abstract basic constraints

$$\mathcal{ABC} = \left\{ c \mid \begin{array}{l} c = (X = t) \quad |t| \leq k \text{ or} \\ c = (X' \neq t') \quad |t'| \leq k, \text{ and } \text{Var}(t') \cap W = \emptyset \end{array} \right\}$$

Note that $\text{Var}(t') \cap W = \emptyset$ expresses the fact that inequalities between variables and cut terms are not allowed. The domain of abstract constraints is defined as follows,

Definition 11.

$$\mathcal{ANC} = \left\{ c \mid \begin{array}{l} c \text{ is a constraint in normal form built with} \\ \text{the logical connectives } \vee, \wedge, \forall \text{ and } \exists \text{ on } \mathcal{ABC} \end{array} \right\}$$

The concepts of abstract constrained atoms and partial abstract interpretations are defined as expected.

Definition 12. An abstract constrained atom is a pair $c|A$ such that $c \in \mathcal{ANC}$ and c is a \mathcal{H} – solvable constraint, A is an atom and $\text{Var}(c) \subseteq \text{Var}(A)$. With \mathcal{B}^a we intend the set of abstract constrained atoms.

The abstract domain is the set of partial interpretations on abstract constrained atoms. A *partial abstract constrained interpretation* for a program, is a pair of set of abstract constrained atoms, $I^a = \langle I^{a+}, I^{a-} \rangle$, not necessary consistent. We consider $\mathcal{I}^a = \{I^a \mid I^a \text{ is a partial interpretation}\}$.

With respect to the case of definite logic programs [5], we need to define a different order on the abstract constraint domain.

This is because the result c^a of an abstract *and* operation on the abstract constraint domain will be an approximation of the result c of the concrete *and* operation on the concrete constraint domain, in the sense that c^a will be “more general” than the abstraction of c (where here “more general” means “is implied under \mathcal{H} ”).

This motivates the definition of the following relation on the abstract constraint domain.

Definition 13. Let $c, c' \in \mathcal{ANC}$. $c \preceq_a c'$ if $\mathcal{H} \models c \rightarrow c'$.

We consider the order \leq_a induced by the preorder \preceq_a , namely the order obtained considering the classes modulo the equivalence induced by \preceq_a .

We define the downward closure of a pair of sets w.r.t. the \leq_a order,

Definition 14. Consider a pair of sets of constrained atoms B .

By $\downarrow B$ we denote the downward closure of $\langle B^+, B^- \rangle$.

$c|A \in \downarrow B^+$ if there exists $c'|A \in B^+$ and $c \leq_a c'$,

$c|A \in \downarrow B^-$ if there exists $c'|A \in B^-$ and $c \leq_a c'$.

Intuitively, a set of constrained atoms I is less or equal than J , if $\downarrow I \subseteq \downarrow J$.

Definition 15. Consider $I, J \in \mathcal{I}^a$.

$I^a \preceq J^a \leftrightarrow$ for all $c|A \in I^{a+} \exists c'|A \in J^{a+}$ such that $c \leq_a c'$ and
for all $c|A \in I^{a-} \exists c'|A \in J^{a-}$ such that $c \leq_a c'$

It is immediate to see that \preceq defines a preorder. We consider the order \leq induced by the preorder \preceq , namely the order obtained by considering the classes \mathbb{I}^a modulo the equivalence induced by \preceq . Then our abstract domain is (\mathbb{I}^a, \leq) . Since the operations on the equivalence classes are independent on the choice of the representative, we denote the class of an interpretation I^a by I^a itself. In the rest of the paper, we often abuse notation by denoting by I^a the equivalence class of I^a or the interpretation I^a itself.

4.2 The Abstraction Function

Let us now define the abstraction function. To this aim we first define the function α_c on constraints. The main idea is to define α_c on the basic constraints as follows: an equality $X = t$ is abstracted to $X = \alpha_k(t)$, while an inequality $X \neq t$ is abstracted to $X \neq t$ if $|t| \leq k$ and to *true* otherwise.

We denote by $\Delta(c)$ the constraint c' in normal form and by Δ the sequence of quantified variables of c' , where c is the quantifier-free part of c' .

Definition 16. The $\text{depth}(k)$ constraint abstraction function is the function $\alpha_c : \mathcal{NC} \rightarrow \mathcal{ANC}$:

$\alpha_c(\Delta(c)) = \Delta, \Delta' \alpha_c(c)$ where $\Delta' = \exists Y_1, \exists Y_2, \dots$, and $Y_i \in (W \cap \text{Var}(\alpha_c(c)))$

$\alpha_c(X = t) = (X = \alpha_k(t))$,

$\alpha_c(\text{false}) = \text{false}$,

$\alpha_c(\text{true}) = \text{true}$,

$\alpha_c(X \neq t) = (X \neq t)$ if $|t| \leq k$,

$\alpha_c(X \neq t) = \text{true}$ if $|t| > k$,

$\alpha_c(A \wedge B) = \alpha_c(A) \wedge \alpha_c(B)$,

$\alpha_c(A \vee B) = \alpha_c(A) \vee \alpha_c(B)$.

Note that the first definition means that all the new variables introduced by the cut terms have to be considered existentially quantified.

Example 17 shows an application of α_c .

Example 17. $c = \forall U((H = f(f(T)) \wedge T \neq f(f(U)) \wedge X = f(U)) \vee (H = f(f(T)) \wedge T \neq f(X) \wedge X \neq f(U))), k = 2.$

$$\begin{aligned} \alpha_c(c) &= \alpha_c(\forall U(\\ &\quad (H = f(f(T)) \wedge T \neq f(f(U)) \wedge X = f(U)) \vee \\ &\quad (H = f(f(T)) \wedge T \neq f(X) \wedge X \neq f(U)))) = \\ &\quad \forall U(\\ &\quad (\alpha_c(H = f(f(T))) \wedge \alpha_c(T \neq f(f(U))) \wedge \alpha_c(X = f(U))) \vee \\ &\quad (\alpha_c(H = f(f(T))) \wedge \alpha_c(T \neq f(X)) \wedge \alpha_c(X \neq f(U))) = \\ &\quad \forall U, \exists Q_1, Q_2 ((H = f(Q_1) \wedge \text{true} \wedge X = f(U)) \vee \\ &\quad (H = f(Q_2) \wedge T \neq f(X) \wedge X \neq f(U))) \quad (Q_1, Q_2 \in W). \end{aligned}$$

The abstraction function α is defined by applying α_c to every constraint of the constrained atoms in the concrete interpretation.

Definition 18. Let $\alpha : \mathcal{I} \rightarrow \mathbb{I}^a$: $\alpha = < \alpha^+, \alpha^- >$

$$\alpha^+(I) = \{c|A \mid c'|A \in I^+ \text{ and } \alpha_c(c') = c\},$$

$$\alpha^-(I) = \{c|A \mid c'|A \in I^- \text{ and } \alpha_c(c') = c\}.$$

As a consequence the function γ on (equivalence classes of) sets of abstract constraints is automatically determined as follows:

Definition 19. Let $\gamma : \mathbb{I}^a \rightarrow \mathcal{I}$:

$$\begin{aligned} \gamma(I^a) &= \cup\{I \mid \alpha(I) \leq I^a\} = \\ &\quad \cup\{I \mid \forall c|A \in \alpha^+(I) \exists c'|A \in I^{a^+} \text{ such that } c \leq_a c' \text{ and} \\ &\quad \quad \forall c|A \in \alpha^-(I) \exists c'|A \in I^{a^-} \text{ such that } c \leq_a c'\} = \\ &\quad \cup\{I \mid \downarrow \alpha(I) \subseteq \downarrow I^a\} = \cup\{I \mid \alpha(I) \subseteq \downarrow I^a\} \end{aligned}$$

Lemma 20. α is additive.

Theorem 21. $< \alpha, \gamma >$ is a Galois insertion of (\mathcal{I}, \subseteq) into (\mathbb{I}^a, \leq) .

4.3 α_c is a Congruence w.r.t. the H-Equivalence

As we have already pointed out in Sect. 3, we want to work with \mathcal{H} -equivalence classes of constraints and, for this purpose, we need to be sure that the above defined function α_c on \mathcal{NC} is a congruence w.r.t. the \mathcal{H} -equivalence. This means that if two constraints $c, c' \in \mathcal{NC}$ are \mathcal{H} -equivalent, then also $\alpha_c(c)$ and $\alpha_c(c')$ have to be \mathcal{H} -equivalent.

In order to understand whether two constraints are \mathcal{H} -equivalent, it is useful to state the following result.

Lemma 22. Consider the inequality $X \neq t$. There exist no arbitrary quantified t_1, \dots, t_n , where $t_i \neq t$, such that $X \neq t$ is \mathcal{H} -equivalent to $\wedge_i X \neq t_i$.

This is a consequence of the fact that we consider the models of the theory CET without the DCA axiom.

The previous result, together with the fact that constraints are in false-simplified form, allows us to claim that α_c is a congruence.

Theorem 23. *Let $c, c' \in \mathcal{NC}$. If $\mathcal{H} \models c \leftrightarrow c'$ then $\mathcal{H} \models \alpha_c(c) \leftrightarrow \alpha_c(c')$.*

4.4 The Abstract Fixpoint Operator

We now define the abstract operations that will replace the concrete ones in the definition of the fixpoint abstract operator. We show that the abstract operations are a correct approximation of the concrete operations.

The definition of the abstract *and* operation is not immediate. The example 24 is meant to give some intuition on some problems that may arise.

Example 24. Consider the following two constraints:

$c_1 = (X = f(Z, f(H)) \wedge S = f(a))$ $c_2 = (U \neq X \wedge Y \neq f(S))$ and $k = 2$.

Consider $\alpha_c(c_1) = \exists Q(X = f(Z, Q) \wedge S = f(a))$ $\alpha_c(c_2) = (U \neq X \wedge Y \neq f(S))$. If we now consider the normalized form of $\alpha_c(c_1) \wedge \alpha_c(c_2)$ the resulting constraint is $\exists Q(U \neq f(Z, Q) \wedge Y \neq f(f(a)) \wedge X = f(Z, Q) \wedge S = f(a))$, which is not an abstract constraint according to definition 11.

The problem is that the normalized form of the logical *and* operation on two abstract constraints is not in general an abstract constraint (the depth of the terms involved in equalities and inequalities can be greater than k and it can contain inequalities between variables and cut terms).

This is the reason why we need to define a new \mathcal{M} operator, on the normalized forms of abstract constraints. The \mathcal{M} operator must cut terms deeper than k and replace by *true* all the inequalities which contain a cut term. Intuitively this is because $X \neq t$, where $\text{Var}(t) \cap W \neq \emptyset$, represents, on the concrete domain, an inequality between a variable and a term longer than k . On the abstract domain, such inequalities are abstracted to the constant *true*.

Definition 25. *Let $\mathcal{M} : \mathcal{NC} \rightarrow \mathcal{ANC}$*

$\mathcal{M}(\Delta(c)) = \Delta, \Delta' \mathcal{M}(c)$ where $\Delta' = \exists Y_1, \exists Y_2, \dots$, where $Y_i \in (W \cap \text{Var}(\mathcal{M}(c)))$.

$\mathcal{M}(X = t) = (X = \alpha_k(t))$

$\mathcal{M}(X \neq t) = (X \neq t)$ if $|t| \leq k$ and $\text{Var}(t) \cap W = \emptyset$

$\mathcal{M}(X \neq t) = (\text{true})$ if $|t| > k$ or $\text{Var}(t) \cap W \neq \emptyset$

$\mathcal{M}(A \wedge B) = \alpha_c(A) \wedge \alpha_c(B)$, $\mathcal{M}(A \vee B) = \alpha_c(A) \vee \alpha_c(B)$

As expected, the \mathcal{M} operator is similar to the α_c operator. The only difference is that \mathcal{M} replace by *true* all the inequalities between variables and cut terms. Since \mathcal{ANC} is a subset of \mathcal{NC} , the *Res* form is defined also on the abstract constraints domain.

Definition 26. *Let $c_1, c_2 \in \mathcal{ANC}$*

$c_1 \tilde{\wedge} c_2 = \mathcal{M}(\text{Res}(c_1 \wedge c_2))$, $c_1 \tilde{\vee} c_2 = \mathcal{M}(\text{Res}(c_1 \vee c_2))$,

$\tilde{\neg} c_1 = \mathcal{M}(\text{Res}(\neg c_1))$, $\tilde{\exists} X c_1 = \exists X c_1$, $\tilde{\forall} X c_1 = \forall X c_1$,

It is worth noting that the procedure *Res* on the abstract domain needs to perform the logical *and* on abstract constraints. This means that most of the observations that can be done on the behavior of the abstract *and* operation, concern also the abstract *or* and *not* operations.

Example 27 illustrates the relation between the abstract *and* operation and the abstraction of the concrete *and* operation. For a sake of simplicity, since in this case it does not affect the result, we write the constraint c_1 in the more compact standard disjunctive form rather than of in extended disjunctive form.

Example 27. $c_1 = \forall K((Y = a \wedge U \neq f(f(K))) \vee Z = a)$, $c_2 = (U = f(f(a)))$.
 Consider $k = 1$. $\alpha_c(c_1) = (Y = a \vee Z = a)$, $\alpha_c(c_2) = \exists V U = f(V)$.
 $\alpha_c(c_1) \tilde{\wedge} \alpha_c(c_2) = \exists V((Y = a \wedge U = f(V)) \vee (Z = a \wedge U = f(V)))$.
 $\alpha_c(\text{Res}(c_1 \wedge c_2)) = \exists V(Z = a \wedge U = f(V))$.
 $\mathcal{H} \models \alpha_c(\text{Res}(c_1 \wedge c_2)) \rightarrow \alpha_c(c_1) \tilde{\wedge} \alpha_c(c_2)$

As already pointed out, the abstract *and* gives a more general constraint than the abstraction of the one computed by the concrete *and* and this is the reason why we have defined an approximation order based on implication (under \mathcal{H}) between constraints.

In order to show that the abstract operations are correct, we prove a stronger property.

Theorem 28. *Let $c_1, c_2 \in \mathcal{NC}$.*

$$\begin{aligned} \alpha_c(c_1) \tilde{\wedge} \alpha_c(c_2) &\geq_a \alpha_c(c_1 \wedge^c c_2), & \alpha_c(c_1) \tilde{\vee} \alpha_c(c_2) &\geq_a \alpha_c(c_1 \vee^c c_2), \\ \exists x \alpha_c(c_1) &= \alpha_c(\exists^c x c_1), & \forall x \alpha_c(c_1) &= \alpha_c(\forall^c x c_1). \end{aligned}$$

As shown by example 29, the correctness property does not hold for the version of abstract “not” which we have defined, if we consider general constraints.

Example 29. Consider $c_1 = (X \neq f(f(a)))$ and $k = 1$.
 $\alpha_c(\neg^c(c_1)) = \exists Y X = f(Y)$ which does not implies $\tilde{\neg}(\alpha_c(c_1)) = \text{false}$.

Since the *not* operator is used by the abstract fixpoint operator on “simpler” constraints (the program constraints) only, we are interested in its behavior on conjunctions of equalities between variables and terms only. For this kind of constraints the following result holds.

Lemma 30. *If $c_1 = (\bigwedge_i (X_i = t_i)) \in \mathcal{NC}$, then $\tilde{\neg} \alpha_c(c_1) \geq_a \alpha_c(\neg^c(c_1))$.*

Now that we have defined the abstract constraints domain and the abstract operations, we can define the abstract fixpoint operator.

Definition 31. *Let $\alpha(P)$ be the program obtained by replacing every constraint c in a clause of P by $\alpha_c(c)$.*

The abstract fixpoint operator: $\mathbb{I}^a \rightarrow \mathbb{I}^a$ is defined as follows, $T_P^{\mathcal{B}^a}(I^a) = T_{\alpha(P)}^{\mathcal{B}^a}(\downarrow I^a)$, where the operations are $\tilde{\exists}$, $\tilde{\forall}$, $\tilde{\neg}$ on \mathcal{ANC} and $\tilde{\vee}$, $\tilde{\wedge}$ on $\mathcal{ANC} \times \mathcal{ANC}$.

By definition, $T_P^{\mathcal{B}^a}$ is a congruence respect to the equivalence classes of the abstract domain. Note also that $T_P^{\mathcal{B}^a}$ is monotonic on the (\mathbb{I}^a, \leq) , because $I \leq J$ implies $\downarrow I \subseteq \downarrow J$.

Lemma 32. $T_P^{\mathcal{B}^a}$ is monotonic on the (\mathbb{I}^a, \leq) .

The proof that the abstract operator is correct w.r.t. the concrete one, is based on the correctness of the abstract operations on the abstract constraints domain.

Theorem 33. $\alpha(T_P^{\mathcal{B}^a}(\gamma(I^a))) \leq T_P^{\mathcal{B}^a}(I^a)$. Then $\alpha(lfp(T_P^{\mathcal{B}})) \leq T_P^{\mathcal{B}^a}(I^a)$.

Consider now a k greater than the maximal depth of the terms involved in the constraints of the clauses in the program P . In this case the abstract operator is also optimal.

Theorem 34. $T_P^{\mathcal{B}^a}(I^a) \leq \alpha(T_P^{\mathcal{B}}(\gamma(I^a)))$.

Let us finally discuss termination properties of the dataflow analyses presented in this section. First note that the set of not equivalent (w.r.t. \mathcal{H}) set of constraints belonging to \mathcal{ANC} is finite.

Lemma 35. Assume that the signature of the program has a finite number of function and predicate symbols. Our $depth(k)$ abstraction is ascending chain finite.

4.5 An Example

We now show how the depth- k analysis works on an example. The program of figure 11 computes the union of two sets represented as lists. We denote the equivalence class of $T_P^{\mathcal{B}^a}$ by $T_P^{\mathcal{B}^a}$ itself. All the computed constraints for the predicate $\neg member$ are shown, while concerning the predicate $\neg union$, for a sake of simplicity, we choose to show just a small subset of the computed answer constraints (written in the more compact standard disjunctive form). Therefore, the concretization of the set of answer constraints for $\neg union$ that we present in figure 11 contains some answer constraints computed by the concrete semantics but not all of them.

As expected the set of answer constraints, computed by the abstract fixpoint operator, is an approximation of the answer constraints, computed by the concrete operator, for the predicates $member$, $union$ and $\neg member$. For example, for the predicate $\neg member(X, Y)$, we compute the answer $\forall L(Y \neq [X, L])$ which correctly approximates the concrete answer $\forall L, H, H_1, L_1(Y \neq [X, L] \wedge Y \neq [H, H_1, L_1])$. While the constraint answer $\exists X \forall H_1, L_1 \exists Z_1, Z_2(A = [X, Z_1] \wedge C = [X, Z_2] \wedge B \neq [H_1, L_1])$ for $union(A, B, C)$, approximates the concrete constraint $A = [X, X]$, $C = [X, X, K]$, $B = K$ and B is not a list, computed by the concrete semantics. Note, in fact, that, if the second argument is not a list, the predicate $member$ finitely fails. Let us now consider Marriott and Søndergaard's abstraction for the program P , with a language where the only constant is a (this assumption does not affect the result). Concerning the predicate $union$ with the empty list as first argument, their abstraction computes the following atoms $union([], a, a)$, $union([], [], [])$, $union([], [a], [a])$, $union([], [a, Z_1], [a, Z_2])$, while we obtain the more precise answer $(A = [] \wedge B = C) | union(A, B, C)$.

$P :$

$\text{union}(A, B, C) : -A = [], B = C.$

$\text{union}(A, B, C) : -A = [X, L], C = [X, K], \neg \text{member}(X, B), \text{union}(L, B, K).$

$\text{union}(A, B, C) : -A = [X, L], \text{member}(X, B), \text{union}(L, B, C).$

$\text{member}(X, Y) : -Y = [X, L].$

$\text{member}(X, Y) : -Y = [H, L], \text{member}(X, L).$

Consider now a depth-2 analysis with $Z_i \in W$.

$$\begin{aligned}
 & \exists L(\quad T_P^{\text{ga}^+} \quad Y = [X, L] \quad) | \text{member}(X, Y). \\
 & \exists H, Z_1(\quad Y = [H, Z_1] \quad) | \text{member}(X, Y). \\
 & \quad A = [] \wedge B = C \quad | \text{union}(A, B, C). \\
 & \exists X, L(\quad A = [X] \wedge B = [X, L] \wedge B = C \quad) | \text{union}(A, B, C). \\
 & \exists X, Y, Z_1(\quad A = [X] \wedge B = [Y, Z_1] \wedge B = C \quad) | \text{union}(A, B, C). \\
 & \exists X, H, L, Z_1(\quad A = [X, Z_1] \wedge B = [X, L] \wedge B = C \quad) | \text{union}(A, B, C). \\
 & \exists X, H, Z_1, Z_2(\quad A = [X, Z_1] \wedge B = [H, Z_2] \wedge B = C \quad) | \text{union}(A, B, C). \\
 & \exists X, K \forall H, L(\quad A = [X] \wedge C = [X, K] \wedge B \neq [H, L] \wedge B = K \quad) | \text{union}(A, B, C). \\
 & \exists X \forall H, L \exists Z_1, Z_2(\quad A = [X, Z_1] \wedge C = [X, Z_2] \wedge B \neq [H, L] \quad) | \text{union}(A, B, C). \\
 & \exists X, K \forall L(\quad A = [X] \wedge C = [X, K] \wedge B \neq [X, L] \wedge B = K \quad) | \text{union}(A, B, C). \\
 & \exists X \forall L \exists Z_1, Z_2(\quad A = [X, Z_1] \wedge C = [X, Z_2] \wedge B \neq [X, L] \quad) | \text{union}(A, B, C). \\
 & \mathbf{A \text{ subset of } } T_P^{\text{ga}^-} \text{ (complete for the predicate } \text{member} \text{)} \\
 & \quad \forall H, L(\quad Y \neq [H, L] \quad) | \text{member}(X, Y). \\
 & \quad \forall L(\quad Y \neq [X, L] \quad) | \text{member}(X, Y). \\
 & \forall X, L, K X_1, L_1 \quad ((A \neq [] \wedge A \neq [X, L]) \vee \\
 & \quad (B \neq C \wedge A \neq [X, L]) \vee \\
 & \quad (B \neq C \wedge C \neq [X, K]) \wedge A \neq [X_1, L_1]) \quad | \text{union}(A, B, C). \\
 & \forall X, L, K X_1, L_1, H, L_2 \quad ((A \neq [] \wedge A \neq [X, L]) \vee \\
 & \quad (B \neq C \wedge A \neq [X, L]) \vee \\
 & \quad (B \neq C \wedge C \neq [X, K]) \wedge A \neq [X_1, L_1]) \vee \\
 & \quad (B \neq C \wedge C \neq [X, L] \wedge B \neq [H, L_2]) \vee \\
 & \quad (A \neq [] \wedge C \neq [X, L] \wedge B \neq [H, L_2]) \quad | \text{union}(A, B, C) \\
 & \quad \vdots
 \end{aligned}$$

Fig. 1. Example 1

The atom $union([], [a, Z_1], [a, Z_2])$, in fact, correctly approximates the predicates deeper than k which have a successful behavior, but it has lost the relation between B and C . As a consequence all the other ground atoms for $union$ computed using the atom $union([], [a, Z_1], [a, Z_2])$, are less precise than the ground instances of the atoms computed by our non-ground abstract semantics.

5 Conclusion

Starting from the hierarchy of semantics defined in [10], our aim was to show that well known analysis for logic programs could be extended to normal logic programs. Based on the framework of abstract interpretation [7,8], we have presented a $depth(k)$ analysis which is able to approximate the answer set of normal logic programs.

It is worth noting that our $depth(k)$ analysis, can be easily generalized to constraint logic programs defined on \mathcal{H} , whose program constraints can be conjunctions of equalities and inequalities. In order to deal with constructive negation, in fact, most of the results presented in this paper hold for first order equality constraints. The only exception is lemma [30] (and consequently theorem [33] and theorem [34]), which is true only for conjunctions of equalities. But a more complex definition of the abstract *not* operator can be defined and proven correct on conjunctions of equalities and inequalities constraints. This alternative definition is, however, less precise than the one defined here. As a consequence theorem [33], where the abstract fixpoint operator uses the new abstract *not* operator, still holds for such “extended” logic programs, while it is not the case for theorem [34].

References

1. K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier and The MIT Press, 1990.
2. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19–20:149–197, 1994.
3. D. Chan. Constructive Negation Based on the Completed Database. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int’l Conf. on Logic Programming*, pages 111–125. The MIT Press, 1988.
4. A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer System*, pages 85–99, 1984.
5. M. Comini. *Abstract Interpretation framework for Semantics and Diagnosis of Logic Programs*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1998.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

7. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.
8. P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
9. F. Fages. Constructive negation by pruning. *Journal of Logic Programming*, 32(2):85–118, 1997.
10. F. Fages and R. Gori. A hierarchy of semantics for normal constraint logic programs. In M.Hanus M.Rodriguez-Artalejo, editor, *Proc. Fifth Int'l Conf. on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 1996.
11. M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2:295–312, 1985.
12. R. Giacobazzi. “Optimal” collecting semantics for analysis in a hierarchy of logic program semantics. In C. Puech and R. Reischuk, editors, *Proc. 13th International Symposium on Theoretical Aspects of Computer Science (STACS'96)*, volume 1046 of *Lecture Notes in Computer Science*, pages 503–514. Springer-Verlag, 1996.
13. S. Koppelberg. *Handbook of Boolean Algebras (Vol.I)*. Elsevier Science Publisher B.V.(North Holland), 1989.
14. K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.
15. K. Kunen. Signed Data Dependencies in Logic Programs. *Journal of Logic Programming*, 7(3):231–245, 1989.
16. J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.
17. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987. Second edition.
18. M.J. Maher. Complete axiomatizations of the algebra of finite, rational and infinite trees. In *Third Symp. on Logic in Computer Science*, pages 348–357, 1988.
19. K. Marriott and H. Sondergaard. Bottom-up Dataflow Analysis of Normal Logic Programs. *Journal of Logic Programming*, 13(2 & 3):181–204, 1992.
20. T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.
21. P. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, 1995.

The Correctness of Set-Sharing

Patricia M. Hill¹, Roberto Bagnara^{2*}, and Enea Zaffanella³

¹ School of Computer Studies,
University of Leeds,
Leeds, LS2 9JT, United Kingdom
`hill@scs.leeds.ac.uk`

² Dipartimento di Matematica,
Università degli Studi di Parma, Italy.
`bagnara@prmat.math.unipr.it`

³ Servizio IX Automazione,
Università degli Studi di Modena, Italy.
`zaffanella@elektro.casa.unimo.it`

Abstract. It is important that practical data flow analysers are backed by reliably proven theoretical results. Abstract interpretation provides a sound mathematical framework and necessary generic properties for an abstract domain to be well-defined and sound with respect to the concrete semantics. In logic programming, the abstract domain **Sharing** is a standard choice for sharing analysis for both practical work and further theoretical study. In spite of this, we found that there were no satisfactory proofs for the key properties of commutativity and idempotence that are essential for **Sharing** to be well-defined and that published statements of the safeness property assumed the occur-check. This paper provides a generalisation of the abstraction function for **Sharing** that can be applied to any language, with or without the occur-check. The results for safeness, idempotence and commutativity for abstract unification using this abstraction function are given.

Keywords: abstract interpretation, logic programming, occur-check, rational trees, set-sharing.

1 Introduction

Today, talking about sharing analysis for logic programs is almost the same as talking about the *set-sharing* domain **Sharing** of Jacobs and Langen [8, 9]. Researchers are primarily concerned with extending the domain with linearity, freeness, depth- k abstract substitutions and so on [2, 4, 12, 13, 16]. Key properties such as commutativity and soundness of this domain and its associated abstract operations are normally assumed to hold. The main reason for this is that [9] not only includes a proof of the soundness but also refers the reader to the thesis of Langen [14] for proofs of commutativity and idempotence.

In abstract interpretation, the concrete semantics of a program is approximated by an abstract semantics. In particular, the concrete domain is replaced

* Much of this work was supported by EPSRC grant GR/L19515.

by an abstract domain and each elementary operation on the concrete domain is replaced by a corresponding abstract operation on the abstract domain. Thus, assuming the global abstract procedure mimics the concrete execution procedure, each operation on elements in the abstract domain must produce an approximation of the corresponding operation on corresponding elements in the concrete domain. The key operation in a logic programming derivation is unification (*unify*) and the corresponding operation for an abstract domain is *aunify*.

An important step in standard unification algorithms is the *occur-check* that avoids the generation of infinite data structures. However, in computational terms, it is expensive and it is well known that Prolog implementations by default omit this check. Although standard unification algorithms that include the occur-check produce a substitution that is idempotent, the resulting substitution when the occur-check is omitted, may not be idempotent. In spite of this, most theoretical work on data-flow analysis of logic programming assume the result of *unify* is always idempotent. In particular both [9] and [14] assume in their proofs of soundness that the concrete substitutions are idempotent. Thus their results do not apply to the analysis of all Prolog programs.

If two terms in the concrete domain are unifiable, then *unify* computes the most general unifier (*mg**u*). Up to renaming of variables, an *mg**u* is unique. Moreover a substitution is defined as a *set* of bindings or equations between variables and other terms. Thus, for the concrete domain, the order and multiplicity of elements are irrelevant in both the computation and semantics of *unify*. It is therefore useful that the abstraction of the unification procedure should be unaffected by the order and multiplicity in which it abstracts the bindings that are present in the substitution. Furthermore, from a practical perspective, it is useful if the global abstract procedure can proceed in a different order to the concrete one without affecting the accuracy of the analysis results. Hence, it is extremely desirable that *aunify* is also commutative and idempotent. However, as discussed later in this paper, only a weak form of idempotence has ever been proved while the only previous proof of commutativity [14] is seriously flawed.

As sharing is normally combined with linearity and freeness domains that are not idempotent or commutative, [2, 12] it may be asked why these properties are important for sharing. In answer to this, we observe that the order and multiplicity in which the bindings in a substitution are analysed affects the accuracy of the linearity and freeness domains. It is therefore a real advantage to be able to ignore these aspects as far as the sharing domain is concerned.

This paper provides a generalisation of the abstraction function for **Sharing** that can be applied to any language, with or without the occur-check. The results for safeness, idempotence and commutativity for abstract unification using this abstraction function are given. Detailed proofs of the results stated in this paper are available in [7].

In the next section, the notation and definitions needed for equality and substitutions in the concrete domain are given. In Section 3, we introduce a new concept called *variable-idempotence* that generalises idempotence to allow for rational trees. In Section 4, we recall the definition of **Sharing** and define its

abstraction function, generalised to allow for non-idempotent substitutions. We conclude in Section 5.

2 Equations and Substitutions

2.1 Notation

For a set S , $\#S$ is the cardinality of S , $\wp(S)$ is the powerset of S , whereas $\wp_f(S)$ is the set of all the *finite* subsets of S . The symbol $Vars$ denotes a denumerable set of variables, whereas \mathcal{T}_{Vars} denotes the set of first-order terms over $Vars$ for some given set of function symbols. The set of variables occurring in a syntactic object o is denoted by $vars(o)$.

2.2 Substitutions

If $x \in Vars$ and $s \in \mathcal{T}_{Vars}$, then $x \mapsto s$ is called a *binding*. A substitution is a total function $\sigma: Vars \rightarrow \mathcal{T}_{Vars}$ that is the identity almost everywhere; in other words, the *domain* of σ ,

$$\text{dom}(\sigma) \stackrel{\text{def}}{=} \{ x \in Vars \mid \sigma(x) \neq x \}$$

is finite. If $t \in \mathcal{T}_{Vars}$, we write $t\sigma$ to denote $\sigma(t)$.

Substitutions are denoted by the set of their *bindings*, thus σ is identified with the set $\{ x \mapsto \sigma(x) \mid x \in \text{dom}(\sigma) \}$. The composition of substitutions is defined in the usual way. Thus $\tau \circ \sigma$ is the substitution such that, for all terms t , $(\tau \circ \sigma)(t) = \tau(\sigma(t))$. A substitution is said *circular* if it has the form $\{x_1 \mapsto x_2, \dots, x_{n-1} \mapsto x_n, x_n \mapsto x_1\}$. A substitution is in *rational solved form* if it has no circular subset. The set of all substitutions in rational solved form is denoted by *Subst*.

2.3 Equations

An *equation* is of the form $s = t$ where $s, t \in \mathcal{T}_{Vars}$. *Eqs* denotes the set of all equations.

We are concerned in this paper to keep the results on sharing as general as possible. In particular, we do not want to restrict ourselves to a specific equality theory. Thus we allow for any equality theory T over \mathcal{T}_{Vars} that includes the *basic axioms* denoted by the following schemata.

$$s = s, \tag{1}$$

$$s = t \iff t = s, \tag{2}$$

$$r = s \wedge s = t \implies r = t, \tag{3}$$

$$f(s_1, \dots, s_n) = f(t_1, \dots, t_n) \iff s_1 = t_1, \dots, s_n = t_n. \tag{4}$$

Of course, T can include other axioms. For example, it is usual in logic programming and most implementations of Prolog to assume an equality theory

based on syntactic identity and characterised by the axiom schemata given by Clark [3]. This consists of the basic axioms together with the following:

$$\neg f(s_1, \dots, s_n) = g(t_1, \dots, t_m) \quad (5)$$

$$\forall z \in \text{Vars} \ \forall t \in (\mathcal{T}_{\text{Vars}} \setminus \text{Vars}) : z \in \text{vars}(t) \implies \neg(z = t). \quad (6)$$

The *identity axioms* characterised by the schemata 5 ensure the equality theory is Herbrand and depends only on the syntax. Equality theory for a non-Herbrand domain replaces these axioms by ones that depend instead on the semantics of the domain. Axioms characterised by the schemata 6 are called the *occur-check axioms* and are an essential part of the standard unification procedure in SLD-resolution.

An alternative approach used in some implementations of Prolog, does not require the occur-check axioms. This approach is based on the theory of rational trees [5, 6]. It assumes the basic axioms and the identity axioms together with a set of *uniqueness axioms* [10, 11]. These state that each equation in rational solved form uniquely defines a set of trees. Thus, an equation $z = t$ where $z \in \text{vars}(t)$ and $t \in (\mathcal{T}_{\text{Vars}} \setminus \text{Vars})$ denotes the axiom (expressed in terms of the usual first-order quantifiers [15]):

$$\forall x \in \text{Vars} : (z = t \wedge (x = t\{z \mapsto x\} \implies z = x)).$$

The basic axioms defined by schemata 1, 2, 3, and 4, which are all that are required for the results in this paper, are included in both these theories.

A substitution σ may be regarded as a set of equations $\{x = t \mid x \mapsto t \in \sigma\}$. A set of equations $e \in \wp_f(\text{Eqs})$ is *unifiable* if there is $\sigma \in \text{Subst}$ such that $T \vdash (\sigma \implies e)$. σ is called a *unifier* for e . σ is said to be a *relevant* unifier of e if $\text{vars}(\sigma) \subseteq \text{vars}(e)$. That is, σ does not introduce any new variables. σ is a most general unifier for e if, for every unifier σ' of e , $T \vdash (\sigma' \implies \sigma)$. An mgu, if it exists, is unique up to the renaming of variables. In this paper, $\text{mgu}(e)$ always denotes a relevant unifier of e .

3 Variable-Idempotence

It is usual in papers on sharing analysis to assume that all the substitutions are idempotent. Note that a substitution σ is *idempotent* if, for all $t \in \mathcal{T}_{\text{Vars}}$, $t\sigma\sigma = t\sigma$. However, the sharing domain is just concerned with the variables. So, to allow for substitutions representing rational trees, we generalise idempotence to *variable-idempotence*.

Definition 1. A substitution σ is variable-idempotent if

$$\forall t \in \mathcal{T}_{\text{Vars}} : \text{vars}(t\sigma\sigma) = \text{vars}(t\sigma).$$

The set of all variable-idempotent substitutions is denoted by $V\text{Subst}$.

It is convenient to use the following alternative characterisation of variable-idempotence: A substitution σ is variable-idempotent if and only if,

$$\forall (x \mapsto t) \in \sigma : \text{vars}(t\sigma) = \text{vars}(t).$$

Thus any substitution consisting of a single binding is variable-idempotent. Moreover, all idempotent substitutions are also variable-idempotent.

Example 1. The substitution $\{x \mapsto f(x)\}$ is not idempotent but is variable-idempotent. Also, $\{x \mapsto f(y), y \mapsto z\}$ is not idempotent or variable-idempotent but is equivalent (with respect to some equality theory T) to $\{x \mapsto f(z), y \mapsto z\}$, which is idempotent.

We define the transformation $\mapsto^{\mathcal{S}} \subseteq \text{Subst} \times \text{Subst}$, called \mathcal{S} -transformation, as follows:

$$\frac{(x \mapsto t) \in \sigma \quad (y \mapsto s) \in \sigma \quad x \neq y}{\sigma \mapsto^{\mathcal{S}} (\sigma \setminus \{y \mapsto s\}) \cup \{y \mapsto s[x/t]\}}$$

Any substitution σ can be transformed to a *variable-idempotent substitution* σ' for σ by a finite sequence of \mathcal{S} -transformations. Furthermore, if the substitutions σ and σ' are regarded as equations, then they are equivalent with respect to any equality theory that includes the basic equality axioms. These two statements are direct consequences of Lemmas 1 and 2, respectively.

Lemma 1. *Let T be an equality theory that satisfies the basic equality axioms. Suppose that σ and σ' are substitutions such that $\sigma \mapsto^{\mathcal{S}} \sigma'$. Then, regarding σ and σ' as sets of equations, $T \vdash (\sigma \iff \sigma')$.*

Proof. Suppose that $(x \mapsto t), (y \mapsto s) \in \sigma$ where $x \neq y$ and suppose also $\sigma' = (\sigma \setminus \{y \mapsto s\}) \cup \{y \mapsto s[x/t]\}$. We first show by induction on the depth of the term s that

$$x = t \implies s = s[x/t].$$

Suppose s has depth 1. If s is x , then $s[x/t] = t$ and the result is trivial. If s is a variable distinct from x or a constant, then $s[x/t] = s$ and the result follows from equality Axiom 1. Suppose now that $s = f(s_1, \dots, s_n)$ and the result holds for all terms of depth less than that of s . Then, by the inductive hypothesis, for each $i = 1, \dots, n$,

$$x = t \implies s_i = s_i[x/t].$$

Hence, by Axiom 4,

$$x = t \implies f(s_1, \dots, s_n) = f(s_1[x/t], \dots, s_n[x/t])$$

and hence

$$x = t \implies f(s_1, \dots, s_n) = f(s_1, \dots, s_n)[x/t].$$

Thus, combining this result with Axiom 3, we have

$$\begin{aligned}\{x = t, y = s\} &\implies \{x = t, y = s, s = s[x/t]\} \\ &\implies \{x = t, y = s[x/t]\}.\end{aligned}$$

Similarly, combining this result with Axioms 2 and 3,

$$\begin{aligned}\{x = t, y = s[x/t]\} &\implies \{x = t, y = s[x/t], s = s[x/t]\} \\ &\implies \{x = t, y = s\}.\end{aligned}$$

□

Note that the condition $x \neq y$ in Lemma 1 is necessary. For example, suppose $\sigma = \{x \mapsto f(x)\}$ and $\sigma' = \{x \mapsto f(f(x))\}$. Then we do not have $\sigma' \implies \sigma$.

Lemma 2. *Suppose that, for each $j = 0, \dots, n$:*

$$\sigma_j = \{x_1 \mapsto t_{1,j}, \dots, x_n \mapsto t_{n,j}\},$$

where $t_{j,j} = t_{j,j-1}$ and if $j > 0$, for each $i = 1, \dots, n$, where $i \neq j$, $t_{i,j} = t_{i,j-1}[x_j/t_{j,j-1}]$. Then, for each $j = 0, \dots, n$,

$$\nu_j = \{x_1 \mapsto t_{1,j}, \dots, x_j \mapsto t_{j,j}\}$$

is variable-idempotent and, if $j > 0$, σ_j can be obtained from σ_{j-1} by a sequence of \mathcal{S} -transformations.

Proof. The proof is by induction on j . Since ν_0 is empty, the base case when $j = 0$ is trivial. Suppose, therefore that $1 \leq j \leq n$ and the hypothesis holds for ν_{j-1} and σ_{j-1} . By the definition of ν_j , we have $\nu_j = \{x_j \mapsto t_{j,j-1}\} \circ \nu_{j-1}$. Consider an arbitrary i , $1 \leq i \leq j$. We will show that $\text{vars}(t_{i,j}\nu_j) = \text{vars}(t_{i,j})$.

Suppose first that $i = j$. Then since $t_{j,j} = t_{j,j-1}$, $t_{j,j-1} = t_{j,0}\nu_{j-1}$ and, by the inductive hypothesis, $\text{vars}(t_{j,0}\nu_{j-1}\nu_{j-1}) = \text{vars}(t_{j,0}\nu_{j-1})$, we have

$$\begin{aligned}\text{vars}(t_{j,j}\nu_j) &= \text{vars}(t_{j,0}\nu_{j-1}\nu_{j-1}\{x_j \mapsto t_{j,j}\}) \\ &= \text{vars}(t_{j,0}\nu_{j-1}\{x_j \mapsto t_{j,j}\}) \\ &= \text{vars}(t_{j,j}\{x_j \mapsto t_{j,j}\}) \\ &= \text{vars}(t_{j,j}).\end{aligned}$$

Suppose now that $i \neq j$. Then,

$$\text{vars}(t_{i,j}) = \text{vars}(t_{i,j-1}\{x_j \mapsto t_{j,j-1}\}).$$

and, by the inductive hypothesis, $\text{vars}(t_{i,j-1}\nu_{j-1}) = \text{vars}(t_{i,j-1})$.

If $x_j \notin \text{vars}(t_{i,j-1})$, then

$$\begin{aligned}\text{vars}(t_{i,j}\nu_{j-1}) &= \text{vars}(t_{i,j-1}\{x_j \mapsto t_{j,j-1}\}\nu_{j-1}) \\ &= \text{vars}(t_{i,j-1}\nu_{j-1}) \\ &= \text{vars}(t_{i,j}).\end{aligned}$$

On the other hand, if $x_j \in \text{vars}(t_{i,j-1})$, then

$$\begin{aligned}
 \text{vars}(t_{i,j}\nu_{j-1}) &= \text{vars}(t_{i,j-1}\{x_j \mapsto t_{j,j-1}\}\nu_{j-1}) \\
 &= \text{vars}(t_{i,j-1}\nu_{j-1}) \setminus \{x_j\} \cup \text{vars}(t_{j,j-1}\nu_{j-1}) \\
 &= \text{vars}(t_{i,j-1}) \setminus \{x_j\} \cup \text{vars}(t_{j,j-1}) \\
 &= \text{vars}(t_{i,j-1}\{x_j \mapsto t_{j,j-1}\}) \\
 &= \text{vars}(t_{i,j}).
 \end{aligned}$$

Thus, in both cases,

$$\begin{aligned}
 \text{vars}(t_{i,j}\nu_j) &= \text{vars}(t_{i,j}\nu_{j-1}\{x_j \mapsto t_{j,j-1}\}) \\
 &= \text{vars}(t_{i,j}\{x_j \mapsto t_{j,j-1}\}) \\
 &= \text{vars}(t_{i,j-1}\{x_j \mapsto t_{j,j-1}\}\{x_j \mapsto t_{j,j-1}\}).
 \end{aligned}$$

However, a substitution consisting of a single binding is variable-idempotent. Thus

$$\begin{aligned}
 \text{vars}(t_{i,j}\nu_j) &= \text{vars}(t_{i,j-1}\{x_j \mapsto t_{j,j-1}\}) \\
 &= \text{vars}(t_{i,j}).
 \end{aligned}$$

Therefore, for each $i = 1, \dots, j$, $\text{vars}(t_{i,j}\nu_j) = \text{vars}(t_{i,j})$. It then follows (using the alternative characterisation of variable-idempotence) that ν_j is variable-idempotent. \square

Example 2. Let

$$\sigma_0 = \{x_1 \mapsto f(x_2), x_2 \mapsto g(x_3, x_4), x_3 \mapsto x_1\}.$$

Then

$$\begin{aligned}
 \sigma_1 &= \{x_1 \mapsto f(x_2), x_2 \mapsto g(x_3, x_4), x_3 \mapsto f(x_2)\}, \\
 \sigma_2 &= \{x_1 \mapsto f(g(x_3, x_4)), x_2 \mapsto g(x_3, x_4), x_3 \mapsto f(g(x_3, x_4))\}, \\
 \sigma_3 &= \{x_1 \mapsto f(g(f(g(x_3, x_4)), x_4)), x_2 \mapsto g(f(g(x_3, x_4)), x_4), x_3 \mapsto f(g(x_3, x_4))\}.
 \end{aligned}$$

Note that σ_3 is variable-idempotent and that $T \vdash \sigma_0 \iff \sigma_3$.

4 Set-Sharing

4.1 The Sharing Domain

The **Sharing** domain is due to Jacobs and Langen [8]. However, we use the definition as presented in [1].

Definition 2. (The *set-sharing* lattice.) *Let*

$$SG \stackrel{\text{def}}{=} \{ S \in \wp_f(\text{Vars}) \mid S \neq \emptyset \}$$

and let $SH \stackrel{\text{def}}{=} \wp(SG)$. *The set-sharing lattice is given by the set*

$$SS \stackrel{\text{def}}{=} \{ (sh, U) \mid sh \in SH, U \in \wp_f(\text{Vars}), \forall S \in sh : S \subseteq U \} \cup \{\perp, \top\}$$

ordered by \preceq_{ss} *defined as follows, for each* $d, (sh_1, U_1), (sh_2, U_2) \in SS$:

$$\begin{aligned} \perp &\preceq_{ss} d, \\ d &\preceq_{ss} \top, \\ (sh_1, U_1) &\preceq_{ss} (sh_2, U_2) \iff (U_1 = U_2) \wedge (sh_1 \subseteq sh_2). \end{aligned}$$

It is straightforward to see that every subset of SS has a least upper bound with respect to \preceq_{ss} . Hence SS is a complete lattice.¹

An element sh of SH abstracts the property of sharing in a substitution σ . That is, if σ is idempotent, two variables x, y must be in the same set in sh if some variable, say v occurs in both $x\sigma$ and $y\sigma$. In fact, this is also true for variable-idempotent substitutions although it is shown below that this needs to be generalised for substitutions that are not variable-idempotent. Thus, the definition of the abstraction function α for sharing, requires an ancillary definition for the notion of *occurrence*.

Definition 3. (Occurrence.)

For each $n \in \mathbb{N}$, $\text{occ}_i: \text{Subst} \times \text{Vars} \rightarrow \wp_f(\text{Vars})$ *is defined for each* $\sigma \in \text{Subst}$ *and each* $v \in \text{Vars}$:

$$\begin{aligned} \text{occ}_0(\sigma, v) &\stackrel{\text{def}}{=} \{v\}, & \text{if } v = v\sigma; \\ \text{occ}_0(\sigma, v) &\stackrel{\text{def}}{=} \emptyset, & \text{if } v \neq v\sigma; \\ \text{occ}_n(\sigma, v) &\stackrel{\text{def}}{=} \{ y \in \text{Vars} \mid x \in \text{vars}(y\sigma) \cap \text{occ}_{n-1}(\sigma, v) \}, & \text{if } n > 0. \end{aligned}$$

It follows that, for fixed values of σ *and* v , $\text{occ}_n(\sigma, v)$ *is monotonic and extensive with respect to the index* n . *Hence, as the range of* $\text{occ}_n(\sigma, v)$ *is restricted to the finite set of variables in* σ , *there is an* $\ell = \ell(\sigma, v) \in \mathbb{N}$ *such that* $\text{occ}_\ell(\sigma, v) = \text{occ}_n(\sigma, v)$ *for all* $n \geq \ell$. *Let*

$$\text{occ}!(\sigma, v) \stackrel{\text{def}}{=} \text{occ}_\ell(\sigma, v).$$

Note that if σ is variable-idempotent, then $\text{occ}!(\sigma, v) = \text{occ}_1(\sigma, v)$. Note also that if $v \neq v\sigma$, then $\text{occ}!(\sigma, v) = \emptyset$. Previous definitions for an occurrence operator such as that for sg in [8] have all been for idempotent substitutions. However, when σ is an idempotent substitution, $\text{occ}!(\sigma, v)$ and $sg(\sigma, v)$ are the same for all $v \in \text{Vars}$.

We base the definition of abstraction on the occurrence operator, $\text{occ}!$.

¹ Notice that the only reason we have $\top \in SS$ is in order to turn SS into a lattice rather than a CPO.

Definition 4. (Abstraction.) The concrete domain *Subst* is related to *SS* by means of the abstraction function $\alpha: \wp(\text{Subst}) \times \wp_f(\text{Vars}) \rightarrow \text{SS}$. For each $\Sigma \in \wp(\text{Subst})$ and each $U \in \wp_f(\text{Vars})$,

$$\alpha(\Sigma, U) \stackrel{\text{def}}{=} \bigsqcup_{\sigma \in \Sigma} \alpha(\sigma, U),$$

where $\alpha: \text{Subst} \times \wp_f(\text{Vars}) \rightarrow \text{SS}$ is defined, for each $\sigma \in \text{Subst}$ and each $U \in \wp_f(\text{Vars})$, by

$$\alpha(\sigma, U) \stackrel{\text{def}}{=} \left(\{ \text{occl}(\sigma, v) \cap U \mid v \in \text{Vars} \} \setminus \{\emptyset\}, U \right).$$

The following result states that the abstraction for a substitution σ is the same as the abstraction for a variable-idempotent substitution for σ .

Lemma 3. Let σ be a substitution, σ' a substitution obtained from σ by a sequence of \mathcal{S} -transformations, U a set of variables and $v \in \text{Vars}$. Then

$$v = v\sigma \iff v = v\sigma', \quad \text{occl}(\sigma, v) = \text{occl}(\sigma', v), \quad \text{and} \quad \alpha(\sigma, U) = \alpha(\sigma', U).$$

Proof. Suppose first that σ' is obtained from σ by a single \mathcal{S} -transformation. Thus we can assume that $x \mapsto t$ and $y \mapsto s$ are in σ where $x \in \text{vars}(s)$ and that

$$\sigma' = (\sigma \setminus \{y \mapsto s\}) \cup \{y \mapsto s[x/t]\}.$$

It follows that, since σ is in rational solved form, σ has no circular subset and hence $v = v\sigma \iff v = v\sigma'$. Thus, if $v \neq v\sigma$, then we have $v \neq v\sigma'$ and $\text{occl}(\sigma, v) = \text{occl}(\sigma', v) = \emptyset$.

We now assume that $v = v\sigma = v\sigma'$ and prove that

$$\text{occ}_m(\sigma, v) \subseteq \text{occl}(\sigma', v).$$

The proof is by induction on m . By Definition 3, $\text{occ}_0(\sigma, v) = \text{occ}_0(\sigma', v) = \{v\}$, so that the result holds for $m = 0$. Suppose then that $m > 0$ and that $v_m \in \text{occ}_m(\sigma, v)$. By Definition 3, there exists $v_{m-1} \in \text{vars}(v_m\sigma)$ where $v_{m-1} \in \text{occ}_{m-1}(\sigma, v)$. Hence, by the inductive hypothesis, $v_{m-1} \in \text{occl}(\sigma', v)$. If $v_{m-1} \in \text{vars}(v_m\sigma')$, then, by Definition 3, $v_m \in \text{occl}(\sigma', v)$. On the other hand, if $v_{m-1} \notin \text{vars}(v_m\sigma')$, then $v_m = y$, $v_{m-1} = x$, and $x \in \text{vars}(s)$ (so that $\text{vars}(t) \subseteq \text{vars}(s[x/t])$). However, by hypothesis, $v = v\sigma$, so that $x \neq v$ and $m > 1$. Thus, by Definition 3, there exists $v_{m-2} \in \text{vars}(t)$ such that $v_{m-2} \in \text{occ}_{m-2}(\sigma, v)$. By the inductive hypothesis, $v_{m-2} \in \text{occl}(\sigma', v)$. Since $y \mapsto s[x/t] \in \sigma'$, and $v_{m-2} \in \text{vars}(s[x/t])$, $v_{m-2} \in \text{vars}(y\sigma')$. Thus, by Definition 3, $y \in \text{occl}(\sigma', v)$.

Conversely, we now prove that, for all m ,

$$\text{occ}_m(\sigma', v) \subseteq \text{occl}(\sigma, v).$$

The proof is again by induction on m . As in the previous case, $\text{occ}_0(\sigma', v) = \text{occ}_0(\sigma, v) = \{v\}$, so that the result holds for $m = 0$. Suppose then that $m > 0$ and

that $v_m \in \text{occ}_m(\sigma', v)$. By Definition 3, there exists $v_{m-1} \in \text{vars}(v_m \sigma')$ where $v_{m-1} \in \text{occ}_{m-1}(\sigma', v)$. Hence, by the inductive hypothesis, $v_{m-1} \in \text{occ}!(\sigma, v)$. If $v_m \in \text{occ}(\sigma, v_{m-1})$, then, by Definition 3, $v_m \in \text{occ}!(\sigma, v)$. On the other hand, if $v_{m-1} \notin \text{vars}(v_m \sigma)$, then $v_m = y$, $v_{m-1} \in \text{vars}(t)$ and $x \in \text{vars}(s)$. Thus, as $y \mapsto s \in \sigma$, $y \in \text{vars}(x\sigma)$. However, since $x \mapsto t \in \sigma$, $v_{m-1} \in \text{vars}(x\sigma)$ so that, by Definition 3, $x \in \text{occ}!(\sigma, v)$. Thus, again by Definition 3, $y \in \text{occ}!(\sigma, v)$.

Thus, if σ' is obtained from σ by a single \mathcal{S} -transformation, we have the required results: $v = v\sigma \iff v = v\sigma'$, $\text{occ}!(\sigma, v) = \text{occ}!(\sigma', v)$, and $\alpha(\sigma, U) = \alpha(\sigma', U)$.

Suppose now that there is a sequence $\sigma = \sigma_1, \dots, \sigma_n = \sigma'$ such that, for $i = 2, \dots, n$, σ_i is obtained from σ_{i-1} by a single \mathcal{S} -step. If $n = 1$, then $\sigma = \sigma'$. If $n > 1$, we have by the first part of the proof that, for each $i = 2, \dots, n$, $v = v\sigma_{i-1} \iff v = v\sigma_i$, $\text{occ}!(\sigma_{i-1}, v) = \text{occ}!(\sigma_i, v)$, and $\alpha(\sigma_{i-1}, U) = \alpha(\sigma_i, U)$, and hence the required results. \square

Example 3. Consider again Example 2. Then

$$\begin{aligned}\text{occ}_1(\sigma_0, x_4) &= \{x_2, x_4\}, \\ \text{occ}_2(\sigma_0, x_4) &= \{x_1, x_2, x_4\}, \\ \text{occ}_3(\sigma_0, x_4) &= \{x_1, x_2, x_3, x_4\} = \text{occ}!(\sigma_0, x_4),\end{aligned}$$

and

$$\text{occ}_1(\sigma_3, x_4) = \{x_1, x_2, x_3, x_4\} = \text{occ}!(\sigma_3, x_4).$$

Thus, if $V = \{x_1, x_2, x_3, x_4\}$,

$$\alpha(\sigma_0, V) = \alpha(\sigma_3, V) = \{\{x_1, x_2, x_3, x_4\}\}.$$

4.2 Abstract Operations for Sharing Sets

We are concerned in this paper in establishing results for the abstract operation *aunify* which is defined for arbitrary sets of equations. However, by building the definition of *aunify* in three steps via the definitions of *amgu* (for sharing sets) and *Amgu* (for sharing domains) and stating corresponding results for each of them, we provide an outline for the overall method of proof for the *aunify* results. Details of all proofs are available in [7].

In order to define the abstract operation *amgu* we need some ancillary definitions.

Definition 5. (Auxiliary functions.) *The closure under union function (also called star-union), $(\cdot)^*: SH \rightarrow SH$, is, for each $sh \in SH$,*

$$sh^* \stackrel{\text{def}}{=} \{S \in SG \mid \exists n \geq 1. \exists T_1, \dots, T_n \in sh. S = T_1 \cup \dots \cup T_n\}.$$

For each $sh \in SH$ and each $T \in \wp_f(\text{Vars})$, the extraction of the relevant component of sh with respect to T is encoded by the function $\text{rel}: \wp_f(\text{Vars}) \times SH \rightarrow SH$ defined as

$$\text{rel}(T, sh) \stackrel{\text{def}}{=} \{S \in sh \mid S \cap T \neq \emptyset\}.$$

For each $sh_1, sh_2 \in SH$, the binary union function $\text{bin}: SH \times SH \rightarrow SH$ is given by

$$\text{bin}(sh_1, sh_2) \stackrel{\text{def}}{=} \{S_1 \cup S_2 \mid S_1 \in sh_1, S_2 \in sh_2\}.$$

The function $\text{proj}: SH \times \wp_f(\text{Vars}) \rightarrow SH$ projects an element of SH onto a set of variables of interest: if $sh \in SH$ and $V \in \wp_f(\text{Vars})$, then

$$\text{proj}(sh, V) \stackrel{\text{def}}{=} \{S \cap V \mid S \in sh, S \cap V \neq \emptyset\}.$$

Definition 6. (*amgu.*) The function *amgu* captures the effects of a binding $x \mapsto t$ on an SH element. Let x be a variable and t a term. Let also $sh \in SH$ and

$$\begin{aligned} A &\stackrel{\text{def}}{=} \text{rel}(\{x\}, sh), \\ B &\stackrel{\text{def}}{=} \text{rel}(\text{vars}(t), sh). \end{aligned}$$

Then

$$\text{amgu}(sh, x \mapsto t) \stackrel{\text{def}}{=} (sh \setminus (A \cup B)) \cup \text{bin}(A^*, B^*).$$

Then we have the following soundness result for *amgu*.

Lemma 4. Let $(sh, U) \in SS$ and $\{x \mapsto t\}, \sigma, \nu \in \text{Subst}$ such that ν is a relevant unifier of $\{x\sigma = t\sigma\}$ and $\text{vars}(x), \text{vars}(t), \text{vars}(\sigma) \subseteq U$. Then

$$\alpha(\sigma, U) \preceq_{ss} (sh, U) \implies \alpha(\nu \circ \sigma, U) \preceq_{ss} (\text{amgu}(sh, x \mapsto t), U).$$

To prove this, observe that, by Lemma 2, if σ is not variable-idempotent, it can be transformed to a variable-idempotent substitution σ' . Hence, by Lemma 3, $\alpha(\sigma, U) = \alpha(\sigma', U)$. Therefore, the proof, which is given in [7], deals primarily with the case when σ is variable-idempotent.

Since a relevant unifier of e is a relevant unifier of any other set e' equivalent to e wrt to the equality theory T , this lemma shows that it is safe for the analyser to perform part or all of the concrete unification algorithm before computing *amgu*.

The following lemmas, proved in [7], show that *amgu* is commutative and idempotent.

Lemma 5. Let $sh \in SH$ and $\{x \mapsto r\} \in \text{Subst}$. Then

$$\text{amgu}(sh, x \mapsto r) = \text{amgu}(\text{amgu}(sh, x \mapsto r), x \mapsto r).$$

Lemma 6. Let $sh \in SH$ and $\{x \mapsto r\}, \{y \mapsto t\} \in \text{Subst}$. Then

$$\text{amgu}(\text{amgu}(sh, x \mapsto r), y \mapsto t) = \text{amgu}(\text{amgu}(sh, y \mapsto t), x \mapsto r).$$

4.3 Abstract Operations for Sharing Domains

The definitions and results of Subsection 4.2 can be lifted to apply to sharing domains.

Definition 7. (Amgu.) *The operation $\text{Amgu}: SS \times \text{Subst} \rightarrow SS$ extends the SS description it takes as an argument, to the set of variables occurring in the binding it is given as the second argument. Then it applies amgu :*

$$\begin{aligned} &\text{Amgu}((sh, U), x \mapsto t) \\ &\stackrel{\text{def}}{=} \left(\text{amgu}\left(sh \cup \{ \{u\} \mid u \in \text{vars}(x \mapsto t) \setminus U \}, x \mapsto t \right), U \cup \text{vars}(x \mapsto t) \right). \end{aligned}$$

The results for amgu can easily be extended to apply to Amgu .

Definition 8. (aunify.) *The function $\text{aunify}: SS \times \text{Eqs} \rightarrow SS$ generalises Amgu to a set of equations e : If $(sh, U) \in SS$, x is a variable, r is a term, $s = f(s_1, \dots, s_n)$ and $t = f(t_1, \dots, t_n)$ are non-variable terms, and $\bar{s} = \bar{t}$ denote the set of equations $\{s_1 = t_1, \dots, s_n = t_n\}$, then*

$$\text{aunify}((sh, U), \emptyset) \stackrel{\text{def}}{=} (sh, U),$$

if $e \in \wp_f(\text{Eqs})$ is unifiable,

$$\begin{aligned} \text{aunify}((sh, U), e \cup \{x = r\}) &\stackrel{\text{def}}{=} \text{aunify}(\text{Amgu}(sh, U), x \mapsto r, e \setminus \{x = r\}), \\ \text{aunify}((sh, U), e \cup \{s = x\}) &\stackrel{\text{def}}{=} \text{aunify}((sh, U), (e \setminus \{s = x\}) \cup \{x = s\}), \\ \text{aunify}((sh, U), e \cup \{s = t\}) &\stackrel{\text{def}}{=} \text{aunify}((sh, U), (e \setminus \{s = t\}) \cup \bar{s} = \bar{t}), \end{aligned}$$

and, if e is not unifiable,

$$\text{aunify}((sh, U), e) \stackrel{\text{def}}{=} \perp.$$

For the distinguished elements \perp and \top of SS

$$\text{aunify}(\perp, e) \stackrel{\text{def}}{=} \perp, \quad \text{aunify}(\top, e) \stackrel{\text{def}}{=} \top.$$

As a consequence of this and the generalisation of Lemmas 4, 5 and 6 to Amgu , we have the following soundness, commutativity and idempotence results required for aunify to be sound and well-defined. As before, the proofs of these results are in [7].

Theorem 1. *Let $(sh, U) \in SS$, $\sigma, \nu \in \text{Subst}$, and $e \in \wp_f(\text{Eqs})$ be such that $\text{vars}(\sigma) \subseteq U$ and ν is a relevant unifier of e . Then*

$$\alpha(\sigma, U) \preceq_{ss} (sh, U) \implies \alpha(\nu \circ \sigma, U) \preceq_{ss} \text{aunify}((sh, U), e).$$

Theorem 2. *Let $(sh, U) \in SS$ and $e \in \wp_f(Eqs)$. Then*

$$\text{aunify}((sh, U), e) = \text{aunify}(\text{aunify}((sh, U), e), e).$$

Theorem 3. *Let $(sh, U) \in SS$ and $e_1, e_2 \in \wp_f(Eqs)$. Then*

$$\text{aunify}(\text{aunify}((sh, U), e_1), e_2) = \text{aunify}(\text{aunify}((sh, U), e_2), e_1).$$

5 Discussion

The SS domain which was first defined by Langen [14] and published by Jacobs and Langen [8] is an important domain for sharing analysis. In this paper, we have provided a framework for analysing non-idempotent substitutions and presented results for soundness, idempotence and commutativity of *aunify*. In fact, most researchers concerned with analysing sharing and related properties using the SS domain, assume these properties hold. Why therefore are the results in this paper necessary? Let us consider each of the above properties one at a time.

5.1 Soundness

We have shown that, for any substitution σ over a set of variables U , the abstraction $\alpha(\sigma, U) = (sh, U)$ is unique (Lemma 3) and the *aunify* operation is sound (Theorem 1). Note that, in Theorem 1, there are no restrictions on σ ; it can be non-idempotent, possibly including cyclic bindings (that is, bindings where the domain variable occurs in its co-domain). Thus this result is widely applicable.

Previous results on sharing have assumed that substitutions are idempotent. This is true if equality is syntactic identity and the implementation uses a unification algorithm based on that of Robinson [17] which includes the occur-check. With such algorithms, the resulting unifier is both unique and idempotent. Unfortunately, this is not what is implemented by most Prolog systems.

In particular, if the algorithm is as described in [11] and used in Prolog III [5], then the resulting unifier is in rational solved form. This algorithm does not generate idempotent or even variable-idempotent substitutions even when the occur-check would never have succeeded. However, it has been shown that the substitution obtained in this way uniquely defines a system of rational trees [5]. Thus our results show that its abstraction using α , as defined in this paper, is also unique and that *aunify* is sound.

Alternatively, if, as in most commercial Prolog systems, the unification algorithm is based on the Martelli-Montanari algorithm, but omits the occur check step, then the resulting substitution may not be idempotent. Consider the following example.

Suppose we are given as input the equation $p(z, f(x, y)) = p(f(z, y), z)$ with an initial substitution that is empty. We apply the steps in Martelli-Montanari procedure but without the occur-check:

	equations	substitution
1	$p(z, f(x, y)) = p(f(z, y), z)$	\emptyset
2	$z = f(z, y), f(x, y) = z$	\emptyset
3	$f(x, y) = f(z, y)$	$\{z \mapsto f(z, y)\}$
4	$x = z, y = y$	$\{z \mapsto f(z, y)\}$
5	$y = y$	$\{z \mapsto f(z, y), x \mapsto z\}$
6	\emptyset	$\{z \mapsto f(z, y), x \mapsto z\}$

Note that we have used three kinds of steps here. In lines 1 and 3, neither argument of the selected equation is a variable. In this case, the outer non-variable symbols (when, as in this example, they are the same) are removed and new equations are formed between the corresponding arguments. In lines 2 and 4, the selected equation has the form $v = t$, where v is a variable and t is not identical to v , then every occurrence of v is replaced by t in all the remaining equations and the range of the substitution. $v \mapsto t$ is then added to the substitution. In line 5, the identity is removed.

Let $\sigma = \{z \mapsto f(z, y), x \mapsto z\}$, be the computed substitution. Then, we have

$$\begin{aligned} vars(x\sigma) &= vars(z) = \{z\}, \\ vars(x\sigma^2) &= vars(f(z, y)) = \{y, z\}. \end{aligned}$$

Hence σ is not variable-idempotent.

We conjecture that the resulting substitution is still unique (up to variable renaming). In this case our results can be applied so that its abstraction using α , as defined in this paper, is also unique and *aunify* is sound.

5.2 Idempotence

Definition 8 defines *aunify* inductively over a set of equations, so that it is important for this definition that *aunify* is both idempotent and commutative.

The only previous result concerning the idempotence of *aunify* is given in thesis of Langen [14, Theorem 32]. However, the definition of *aunify* in [14] includes the renaming and projection operations and, in this case, only a weak form of idempotence holds. In fact, for the basic *aunify* operation as defined here and without projection and renaming, idempotence has never before been proven.

5.3 Commutativity

In the thesis of Langen the “proof” of commutativity of *amgu* has a number of omissions and errors [14, Lemma 30]. We highlight here, one error which we were unable to correct in the context of the given proof.

To make it easier to compare, we adapt our notation and, define *amge* only in the case that a is a variable:

$$amge(a, b, sh) \stackrel{\text{def}}{=} amgu(sh, a \mapsto b).$$

To prove the lemma, it has to show that:

$$\text{amge}(a_2, b_2 \text{ amge}(a_1, b_1, sh)) = \text{amge}(a_1, b_1, \text{amge}(a_2, b_2, sh)).$$

holds when a_1 and a_2 are variables. This corresponds to “the second base case” of the proof. We use Langen’s terminology:

- A set of variables X is at a term t iff $\text{var}(t) \cap X \neq \emptyset$.
- A set of variables X is at i iff X is at a_i or b_i .
- A union $X \cup_i Y$ is of Type i iff X is at a_i and Y is at b_i .

Let $\text{lhs} \stackrel{\text{def}}{=} \text{amge}(a_2, b_2, \text{amge}(a_1, b_1, S))$, and $\text{rhs} \stackrel{\text{def}}{=} \text{amge}(a_1, b_1, \text{amge}(a_2, b_2, S))$.

Let also $Z \in \text{lhs}$ and $T \stackrel{\text{def}}{=} \text{aunify}(a_1, b_1, S)$. Consider the case when

$$\begin{aligned} Z &= X \cup_2 Y \text{ where } X \in \text{rel}(a_2, T), Y \in \text{rel}(b_2, T), \\ X &= U \cup_1 V \text{ where } U \in \text{rel}(a_1, sh), V \in \text{rel}(b_1, sh) \end{aligned}$$

and $U \cap (\text{vars}(a_2) \cup \text{vars}(b_2)) = \emptyset$ (that is, U is not at 2). Then the following quote [14, page 53, line 23] applies:

In this case $(U \cup_1 V) \cup_2 Y = U \cup_1 (V \cup_2 Y)$. By the inductive assumption $V \cup_2 Y$ is in the rhs and therefore so is Z .

We give a counter-example to the statement “ $V \cup_2 Y$ is in the rhs”.

Suppose a_1, b_1, a_2, b_2 are variables. We let each of a_1, b_1, a_2, b_2 denote both the actual variable and the singleton set containing that variable. Suppose $sh = \{a_1, b_1 a_2, b_2\}$. Then, from the definition of amge ,

$$\text{lhs} = \{a_1 b_1 a_2 b_2\}, \quad \text{rhs} = \{a_1 b_1 a_2 b_2\}, \quad T = \{a_1 b_1 a_2, b_2\}.$$

Let $Z = a_1 b_1 a_2 b_2$, $X = a_1 b_1 a_2$, $Y = b_2$, $U = a_1$, $V = b_1 a_2$. All the above conditions. However $V \cup_2 Y = b_1 a_2 b_2$ and this is not in $\{a_1 b_1 a_2 b_2\}$.

References

1. R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. In P. Van Hentenryck, editor, *Static Analysis: Proceedings of the 4th International Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 53–67, Paris, France, 1997. Springer-Verlag, Berlin.
2. M. Bruynooghe and M. Codish. Freeness, sharing, linearity and correctness — All at once. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Static Analysis, Proceedings of the Third International Workshop*, volume 724 of *Lecture Notes in Computer Science*, pages 153–164, Padova, Italy, 1993. Springer-Verlag, Berlin. An extended version is available as Technical Report CW 179, Department of Computer Science, K.U. Leuven, September 1993.
3. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322, Toulouse, France, 1978. Plenum Press.

4. M. Codish, D. Dams, G. Filé, and M. Bruynooghe. Freeness analysis for logic programs-and correctness? In D. S. Warren, editor, *Logic Programming: Proceedings of the Tenth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 116–131, Budapest, Hungary, 1993. The MIT Press. An extended version is available as Technical Report CW 161, Department of Computer Science, K.U. Leuven, December 1992.
5. A. Colmerauer. Prolog and Infinite Trees. In K. L. Clark and S. Å. Tärnlund, editors, *Logic Programming, APIC Studies in Data Processing*, volume 16, pages 231–251. Academic Press, New York, 1982.
6. A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'84)*, pages 85–99, Tokyo, Japan, 1984. ICOT.
7. P. M. Hill, R. Bagnara, and E. Zaffanella. The correctness of set-sharing. Technical Report 98.03, School of Computer Studies, University of Leeds, 1998.
8. D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference*, MIT Press Series in Logic Programming, pages 154–165, Cleveland, Ohio, USA, 1989. The MIT Press.
9. D. Jacobs and A. Langen. Static analysis of logic programs for independent AND parallelism. *Journal of Logic Programming*, 13(2&3):291–314, 1992.
10. J. Jaffar, J-L. Lassez, and M. J. Maher. Prolog-II as an instance of the logic programming scheme. In M. Wirsing, editor, *Formal Descriptions of Programming Concepts III*, pages 275–299. North Holland, 1987.
11. T. Keisu. *Tree Constraints*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, May 1994. Also available in the SICS Dissertation Series: SICS/D–16–SE.
12. A. King. A synergistic analysis for sharing and groundness which traces linearity. In D. Sannella, editor, *Proceedings of the Fifth European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 363–378, Edinburgh, UK, 1994. Springer-Verlag, Berlin.
13. A. King and P. Soper. Depth- k sharing and freeness. In P. Van Hentenryck, editor, *Logic Programming: Proceedings of the Eleventh International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 553–568, Santa Margherita Ligure, Italy, 1994. The MIT Press.
14. A. Langen. *Static Analysis for Independent And-Parallelism in Logic Programs*. PhD thesis, Computer Science Department, University of Southern California, 1990. Printed as Report TR 91-05.
15. M. J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, pages 348–357, Edinburgh, Scotland, 1988. IEEE Computer Society.
16. K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(2&3):315–347, 1992.
17. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing

Valérie Gouranton

IRISA/INRIA, IFSIC

Campus universitaire de Beaulieu, 35042 Rennes Cedex, France

tel : 33 2 99 84 74 85, fax : 33 2 99 84 71 71,

gouranton@irisa.fr

<http://www.irisa.fr/lande/>

Abstract. We consider specifications of analysers expressed as compositions of two functions: a semantic function, which returns a natural semantics derivation tree, and a property defined by recurrence on derivation trees. A recursive definition of a dynamic analyser can be obtained by fold/unfold program transformation combined with deforestation. A static analyser can then be derived by abstract interpretation of the dynamic analyser. We apply our framework to the derivation of a dynamic backward slicing analysis for a logic programming language.

1 Introduction

A large amount of work has been devoted to program analysis during the last two decades, both on the practical side and on the theoretical issues. However, most of the program analysers that have been implemented or reported in the literature so far are concerned with one specific property, one specific language and one specific service (dynamic or static). A few generic tools have been proposed but they are generally restricted to one class of properties or languages, or limited in their level of abstraction. We believe that there is a strong need for environments supporting the design of program analysers and that more effort should be put on the *software engineering* of analysers.

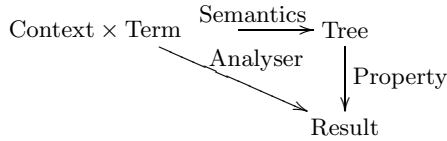
We present a framework for designing analysers from operational specifications by program transformation (folding/unfolding). The analysis specification has two components: a semantics of the programming language and a definition of the property.

The advantage of this two-fold specification is that the definition of the property can be kept separate from the semantics of the programming language. Ideally, properties can be specified in terms of the derivation tree of the operational semantics. Specific analysers can then be obtained systematically by instantiating semantics of the programming language. We focus here on slicing of logic programs. The general approach is detailed in [12].

Natural semantics [7, 14] are a good starting point for the definition of analysers because they are both structural (compositional) and intensional. They are structural because the semantics of a phrase in the programming language

is derived from the semantics of subphrases; they are intensional because the derivation tree that is associated with a phrase in the programming language contains the intermediate results (the semantics of subphrases). These qualities are significant in the context of program analysis because compositionality leads to more tractable proof techniques and intensionality makes it easier to establish the connection between the result of the analysis and its intended use.

Our semantics is defined formally as a function taking a term and an evaluation context and returning a derivation tree. The property itself is a function from derivation trees to a suitable abstract domain. The composition of these two functions defines a dynamic *a posteriori* analysis. It represents a function which initially calculates the trace of a complete execution (a derivation tree) of a program before extracting the required property. Program transformations *via* extended folding/unfolding techniques and simplification rules allow to obtain a recursive definition of the dynamic analyser (which does not call the property function). This function is in fact a dynamic *on the fly* analyser in the sense that it calculates the required property progressively during program execution. The following diagram shows the general organisation:



The key points of the approach proposed here are the following:

- The derivation is achieved in a systematic way by using functional transformations: unfolding and folding.
- It is applicable to a wide variety of languages and properties because it is based on natural semantics definitions.

As mentioned before, some of the analyses that we want to specify are dynamic and others are static. There is no real reason why these two categories of analyses should be seen as belonging to different worlds. In the paper we focus on dynamic analysis, considering that static analysis can be obtained in a second stage as an abstract interpretation of the dynamic analysis as presented in [10]. We outline this derivation in the conclusion. Note that our approach introduces a clear separation between the specification of an analysis (defined as a property on semantics derivation trees) and the algorithm that implements it.

We illustrate the framework by the formal derivation of a slicing analysis for a logic programming language. The different stages of the derivation are detailed in the following sections. Section 2 introduces contexts, terms, derivation trees and the semantics function. The abstract domain and the property function are presented in section 3. The transformation of the composition of the two specification functions (the semantics and the property) into a dynamic *on the fly* analyser is described in section 4. Related work, conclusion and avenues for further research are discussed in section 5 and section 6 respectively.

2 Natural Semantics

The natural semantics of a language is a set of axioms and inference rules that define a relation between a context, a term in the programming language and a result. A natural semantics derivation tree has the form:

$$\text{Proof-Tree} = [\text{RN}] \frac{\text{Proof-Tree}_1 \quad \dots \quad \text{Proof-Tree}_n}{\text{STT}}$$

where RN is the name of the rule used to derive STT. The conclusion STT is a statement, that is to say a triple consisting of a context, a term and a result.

Let **C** be the set of contexts, **T** the type of terms of the language and **PT** the type of derivation trees, we have:

$$\begin{aligned} \text{PT} &= \text{STT} \times (\text{list PT}) \times \text{RN} \\ \text{STT} &= \text{C} \times \text{T} \times \text{NF} \\ \text{T} &= \text{PP} \times \text{I} \end{aligned}$$

Derivation trees are made of a statement (the conclusion), a list of derivation trees (the premises) and the name of rule applied to derive the conclusion. We assume that a term is a pair of a program point and an expression. **STT** denotes the type of statements, **RN** rule names, **NF** normal forms (program results), **PP** program points and **I** expressions.

The Semantics of a Logic Programming Language

We assume a program *Prog* which is a collection of predicate definitions of the form $[P_k(x_1, \dots, x_n) = B_k]$. The body B_k of a predicate is in normal form and it contains only variables from $\{x_1, \dots, x_n\}$. Normal forms are first order formulae (also called “goal formulae” in [16]) built up from predicate applications using only the connectives “and”, “or”, and “there exists”. Their syntax is defined by:

$$I ::= \text{Op}(x_1, x_2, x_3) \mid x = t \mid U_1 \wedge U_2 \mid U_1 \vee U_2 \mid \exists x. U_1 \mid P_k(y_1, \dots, y_n)$$

where **Op** stands for basic predicates [1] and P_k for user-defined predicates. U_i are terms of type **T**. We assume that each variable x occurring in a term $\exists x. U_1$ is unique. In a program, each subterm in this syntax is associated with a program point (using pairs).

As an illustration of this syntax, Figure [1] presents a small program in a logic programming syntax and shows its translation into normal form. Program points are represented by π_i . Note that some program points are omitted for the sake of readability. The program defines two predicates P and Q . The main predicate is Q . The recursive predicate P computes the length n of the list l of integers, the sum sum of the elements of the list, the maximum max and the minimum min of the list l . The average av of the list is computed by the predicate Q via P (the value of sum obtained by P is divided by the length n of the list).

¹ We consider only ternary basic predicates here, but other arities are treated in the same way.

Definition of the program in a logic programming syntax

$$\begin{aligned}
 &P(\text{nil}, 1, 0, 0, 0) \\
 &P((x, \text{nil}), 1, x, x, x) \\
 &P((x, xs), n, \text{sum}, \text{max}, \text{min}) = (\pi_1, P(xs, n', \text{sum}', \text{max}', \text{min}')) \\
 &\quad (\pi_2, \text{Ad}(n', 1, n)) \\
 &\quad (\pi_3, \text{Ad}(\text{sum}', x, \text{sum})) \\
 &\quad (\pi_4, \text{Max}(\text{max}', x, \text{max})) \\
 &\quad (\pi_5, \text{Min}(\text{min}', x, \text{min})) \\
 &Q(l, av, \text{max}, \text{min}) = (\pi_6, P(l, n, \text{sum}, \text{max}, \text{min})) \\
 &\quad (\pi_7, \text{Div}(\text{sum}, n, av))
 \end{aligned}$$

Normal form of the program

$$\begin{aligned}
 P(l, n, \text{sum}, \text{max}, \text{min}) = & ((l = \text{nil}) \wedge (n = 1) \wedge (\text{sum} = 0) \wedge (\text{max} = 0) \wedge (\text{min} = 0)) \vee \\
 & (\exists x. (l = (x, \text{nil})) \wedge (n = 1) \wedge (\text{sum} = x) \wedge (\text{max} = x) \wedge (\text{min} = x)) \vee \\
 & (\exists x. \exists xs. \exists n'. \exists \text{sum}'. \exists \text{max}'. \exists \text{min}'. l = (x, xs) \wedge \\
 & \quad (\pi_1, P(xs, n', \text{sum}', \text{max}', \text{min}')) \wedge \\
 & \quad (\pi_2, \text{Ad}(n', 1, n)) \wedge \\
 & \quad (\pi_3, \text{Ad}(\text{sum}', x, \text{sum})) \wedge \\
 & \quad (\pi_4, \text{Max}(\text{max}', x, \text{max})) \wedge \\
 & \quad (\pi_5, \text{Min}(\text{min}', x, \text{min}))) \\
 Q(l, av, \text{max}, \text{min}) = & (\exists n. \exists \text{sum}. \\
 & \quad (\pi_6, P(l, n, \text{sum}, \text{max}, \text{min})) \wedge \\
 & \quad (\pi_7, \text{Div}(\text{sum}, n, av)))
 \end{aligned}$$

Fig. 1. A simple logic program

Following [15], we assume an infinite set of program variables **Pvar** and an infinite set of renaming variables **Rvar**. Terms and substitutions are constructed using program variables and renaming variables. We distinguish two kinds of substitutions: program variable substitutions (**Subst**) whose domain and co-domain are subsets of **Pvar** and **Rvar** respectively, and renaming variable substitutions (**Rsubst**) whose domain and co-domain are subsets of **Rvar**:

$$\begin{aligned}
 \text{Subst} &= \text{Pvar} \rightarrow \text{Rterm} \\
 \text{Rsubst} &= \text{Rvar} \rightarrow \text{Rterm}
 \end{aligned}$$

where **Rterm** represents a term constructed with renaming variables **Rvar**. By convention, we use $\theta \in \text{Subst}$ for a program variable substitution and $\sigma \in \text{Rsubst}$ for a renaming variable substitution. The definition of substitution composition is modified to take account the role held by renaming variables. The modification occurs when $\theta \in \text{Subst}$ and $\sigma \in \text{Rsubst}$, we have $\sigma \circ \theta \in \text{Subst}$ defined by:

$$\begin{aligned}
 \text{dom}(\sigma \circ \theta) &= \text{dom}(\theta) \\
 (\sigma \circ \theta)(x) &= \sigma(\theta(x)) \text{ for all } x \in \text{dom}(\theta)
 \end{aligned}$$

The domain of contexts for this language is defined by $\mathbf{C} = \mathbf{Tree}(\mathbf{Subst})$ where $\mathbf{Tree}(H)$ is the type of binary trees with leaves of type H . We define contexts as binary trees of substitutions to take into account the non deterministic nature of the language. So, we gather in one derivation the computation of all the substitutions of a program. A particular control strategy for the implementation of the language corresponds to a particular ordering of the leaves of substitutions trees. For instance, the list of results of the usual depth-first evaluation strategy of Prolog is precisely the leaves of the substitution tree produced by our semantics ordered from left to right. We write $N(T_1, T_2)$ for a tree with subtrees T_1 and T_2 .

[Op]	$C \vdash (\pi, \mathbf{Op}(x_1, x_2, x_3)) \rightarrow \overline{\mathbf{Op}}(C, x_1, x_2, x_3)$
[Eq]	$C \vdash (\pi, x = t) \rightarrow \overline{\mathbf{unif}}(C, x, t)$
[\wedge]	$\frac{C \vdash U_1 \rightarrow R_1 \quad R_1 \vdash U_2 \rightarrow R_2}{C \vdash (\pi, U_1 \wedge U_2) \rightarrow R_2}$
[\vee]	$\frac{C \vdash U_1 \rightarrow R_1 \quad C \vdash U_2 \rightarrow R_2}{C \vdash (\pi, U_1 \vee U_2) \rightarrow \mathbf{union}(C, R_1, R_2)}$
[\exists]	$\frac{\overline{\mathbf{Add}}(C, x, rx) \vdash U_1 \rightarrow R_1}{C \vdash (\pi, \exists x. U_1) \rightarrow \overline{\mathbf{Drop}}(R_1, x)} \quad rx \in \mathbf{Rvar} \quad \text{fresh variable}$
[Call]	$\frac{\overline{\mathbf{Ren}}_k(C) \vdash B_k \rightarrow R_1}{C \vdash (\pi, P_k(y_1, \dots, y_n)) \rightarrow \overline{\mathbf{Ext}}_k(C, R_1)} \quad \text{with } [P_k(x_1, \dots, x_n) = B_k] \in \mathbf{Prog}$
$\overline{F}(N(T_1, T_2), x_1, \dots, x_n) = N(\overline{F}(T_1, x_1, \dots, x_n), \overline{F}(T_2, x_1, \dots, x_n))$ $\overline{F}(\theta, x_1, \dots, x_n) = F(\theta, x_1, \dots, x_n)$	
$\mathbf{op}(\theta, x_1, x_2, x_3) = \mathbf{let} \ \sigma = [(\theta(x_1) \ \mathbf{op} \ \theta(x_2))/\theta(x_3)] \ \mathbf{in} \ \sigma \circ \theta$ $\quad \text{if } \theta(x_1) \text{ and } \theta(x_2) \text{ are ground and } \theta(x_3) \in \mathbf{Rvar}, \ \perp \text{ otherwise}$	
$\mathbf{unif}(\theta, x, t) = \mathbf{let} \ \sigma = \mathbf{mgu}(\theta(x), \theta(t)) \ \mathbf{in} \ \sigma \circ \theta \quad \text{if } \theta(x) \text{ and } \theta(t) \text{ can be unified,}$ $\quad \perp \text{ otherwise}$	
$\mathbf{union}(N(T_1, T_2), N(U_1, U_2), N(V_1, V_2)) = N(\mathbf{union}(T_1, U_1, V_1), \mathbf{union}(T_2, U_2, V_2))$	
$\mathbf{union}(\theta, U, V) = N(U, V)$	
$\mathbf{Add}(\theta, pv, rv) = \theta[rv/pv] \quad \text{with } v \neq pv \Rightarrow \theta[rv/pv](v) = \theta(v) \quad \text{and } \theta[rv/pv](pv) = rv$	
$\mathbf{Drop}(\theta, pv) = \theta_{/pv} \quad \text{with } v \neq pv \Rightarrow \theta_{/pv}(v) = \theta(v) \quad \text{and } \theta_{/pv}(pv) = \perp$	
$\mathbf{Ren}_k(\theta) = [\theta(y_i)/x_i]$	
$\mathbf{Ext}_k(\theta, \theta') = \sigma \circ \theta \quad \text{with } \theta' = \sigma \circ [\theta(y_i)/x_i]$	

Fig. 2. Natural semantics of a logic programming language

The natural semantics of a simple logic programming language using the usual inference rule presentation is presented in Figure 2. The normal forms calculated by the rules are contexts. In the figure, $\overline{F}(T)$ denotes the application of a function F to all the substitutions of a tree T and its result is also a tree. The function op represents the interpretation of operator Op . The substitution $unif(\theta, x, t)$ of **Subst** is defined for the unification of x and t via θ (rule **Eq**). The rule \wedge is not surprising, the first formula U_1 of the conjunction is evaluated and the result R_1 is taken as the new context for the evaluation of the second formula U_2 of the conjunction; the result R_2 is the final result. For the rule \vee , the subtrees corresponding to the sub-formulae of the disjunction are evaluated independently. The function $union(T_1, T_2, T_3)$ is needed to build a new substitution tree joining the trees T_2 and T_3 produced by two subgoals. Its first argument is the initial substitution, which is used to identify the points where the joins have to be introduced (these points are the leaves of T_1). The argument θ as the initial substitution can be ignored because substitutions are added to contexts, generating new contexts. The rule \exists uses two functions *Add* and *Drop*. *Add* is used to add a program variable in a substitution (the new program variable is attached to a free renaming variable) and *Drop* removes a variable from a substitution.

For the rule **Call**, two definitions of substitutions are needed. $Ren_k(C)$ creates a new substitution to execute the body of a clause (it amounts to a variable renaming) because the body B_k of a clause contains formal parameters x_i and C contains program variables y_i . $Ext_k(C, R_1)$ propagates the result of a predicate in the calling substitutions because C contains variables y_i and R_1 contains formal parameters x_i . From the definition of Ren_k , we see that the body B_k of a predicate is evaluated in an environment defining exactly the formal parameters of the predicate P_k .

The formal definitions of the functions introduced informally before are presented in the bottom of Figure 2.

In order to make formal manipulations easier, we express the construction of natural semantics derivation trees in a functional framework. The semantic function S is a partial function of type:

$$C \times T \rightarrow PT$$

The important issue about the type of the semantic function is that it returns the whole natural semantics derivation tree, rather than just the result of the program. This choice makes it easier to define intensional analyses. The fact that we describe the semantics in a functional framework does not prevent us from dealing with non deterministic languages, as we show for a logic programming language. This is because we can use **NF** and **C** to represent sets of possible results and contexts.

We use the notation $X.ty$ to denote the field of type **TY** of X . For example, we will make intensive use of the following expressions in the rest of the paper:

	type	meaning
$PT.stt$	STT	conclusion of PT
$PT.lpt$	(list PT)	premisses of PT
$PT.rn$	RN	name of the rule used at the root of PT
$PT.stt.c$	C	context of the conclusion sequent of PT
$PT.stt.t.i$	I	term of the conclusion sequent of PT
$PT.stt.t.pp$	PP	program point of the conclusion sequent of PT
$PT.stt.nf$	NF	normal form of the conclusion sequent of PT

The semantics in functional form is presented in Figure 3. The semantics function of Figure 3 takes two arguments (the context C and the term T) and it returns a derivation tree. The derivation tree contains the conclusion $(C, T, \mathcal{F}^k(C, \bar{R}, E))$ of type STT, where \mathcal{F}^k is the result of the program in functional form, the list of subtrees and the name k of the rule used to derive the conclusion. The body of the function is a list of cases selected by pattern matching on the form of the term. The function is defined by recurrence on the term. The set of definitions Prog is used as an implicit parameter of the semantics.

```

 $\mathcal{S}(C, T) = \text{case } T \text{ of}$ 
   $(\pi, \text{Op}(x_1, x_2, x_3)) : ((C, T, \overline{\text{Op}}(C, x_1, x_2, x_3)), \text{nil}, \text{Op})$ 
   $(\pi, \text{Eq}(x, t)) : ((C, T, \overline{\text{unif}}(C, x, t)), \text{nil}, \text{Eq})$ 
   $(\pi, \text{And}(U_1, U_2)) : \text{let } PT_1 = \mathcal{S}(C, U_1)$ 
     $R_1 = PT_1.stt.nf$ 
     $PT_2 = \mathcal{S}(R_1, U_2)$ 
     $R_2 = PT_2.stt.nf$ 
  in  $((C, T, R_2), [PT_1, PT_2], \wedge)$ 
   $(\pi, \text{Or}(U_1, U_2)) : \text{let } PT_1 = \mathcal{S}(C, U_1)$ 
     $R_1 = PT_1.stt.nf$ 
     $PT_2 = \mathcal{S}(C, U_2)$ 
     $R_2 = PT_2.stt.nf$ 
  in  $((C, T, \text{union}(C, R_1, R_2)), [PT_1, PT_2], \vee)$ 
   $(\pi, \text{Exists}(x, U_1)) : \text{let } PT_1 = \mathcal{S}(\overline{\text{Add}}(C, x, rx), U_1)$ 
     $R_1 = PT_1.stt.nf$ 
  in  $((C, T, \overline{\text{Drop}}(R_1, x)), [PT_1], \exists)$ 
   $(\pi, \text{Call}(P_k(y_1, \dots, y_n))) : \text{let } PT_1 = \mathcal{S}(\overline{\text{Ren}}_k(C), B_k)$ 
     $R_1 = PT_1.stt.nf$ 
  in  $((C, T, \overline{\text{Ext}}_k(C, R_1)), [PT_1], \text{Call})$ 
  with  $[P_k(x_1, \dots, x_n) = B_k] \in \text{Prog}$ 

```

Fig. 3. The semantics function of a logic programming language

3 Specification of a Slicing Property

Slicing² a program consists in constructing a reduced version of the program (called a program *slice*) containing only those statements that affect a given set of variables at given program points (this set is called *the slicing criterion*). In program debugging, slicing makes it possible for a software engineer to focus on the relevant parts of the code. Slicing is also useful for testing, program understanding and in maintenance activities. Because of this diversity of applications, different variations on the notion of slicing have been proposed, as well as a number of methods to compute slices. First, a program slice can either be *executable* or not. Producing an executable slice makes it possible to apply further treatments to the result of the analysis. Another important distinction is between *static* and *dynamic* slicing. In the first case, the slice is computed without any assumption on the inputs, whereas the latter relies on some specific input data. Slicing algorithms can also be distinguished by their direction. *Backward* slicing identifies the statements of a program that may have some impact on the criterion whereas *forward* slicing returns the statements which may be influenced by the criterion. In this paper, we consider *dynamic backward slicing* with executable slices. Static slicing algorithms can be derived by abstract interpretation of dynamic slicing analysers ; this construction is sketched in the conclusion. We can describe forward slicing analysers in a similar way but slicing analysers producing non executable slices do not fit well into our framework since the specification of the analysis is a relation between the semantics of the original program and the semantics of the slice as presented in [10] .

Slicing was originally proposed by Weiser for imperative languages [29] and its application to logic programming [23] and functional programming [18] have been studied recently. In fact, the concept of slicing itself is very general: it is not tied to one specific style of programming³ and it can lead to dynamic as well as static analysers [25].

A slicing analysis for a logic programming language (with programs in normal form) according to a program point and a set of variables of interest consists in keeping only the sub-goals of disjunctions of each clause (a clause defines a predicate) being able to affect the value of the variables of interest. If all sub-goals of a formula of the disjunction are dropped, then this formula is dropped. If all formulae of the disjunction of goals are dropped, then the clause is dropped. In the opposite case, the head of the clause defining the predicate is kept.

Let us take the program in normal form of Figure 1 to illustrate dynamic backward slicing. We assume that we are interested only in the value of the variable *av* at the program point π_7 . The pair $\{(\pi_7, av)\}$ is called the slicing criterion. The dynamic slice of the program is extracted for one particular input. For instance, if we execute the predicate *Q* with *nil* as the initial value of *l*, we get:

² More precisely “backward slicing”.

³ Even if the details of the resulting analyses are of course.

$$\begin{aligned}
&P(\text{nil}, 1, 0, 0, 0) \\
Q(l, av, max, min) &= (\pi_6, P(l, n, sum, max, min)) \\
&\quad (\pi_7, Div(sum, n, av))
\end{aligned}$$

The predicate P is not recursively called and the first disjunctive part is satisfied, the third clause of P is never executed. The definition of the predicate Q is kept because all its clauses are useful to compute the variable av . If we consider the execution of the program with $(2, (3, \text{nil}))$ as initial value of l , we recursively call the predicate P , we get:

$$\begin{aligned}
&P((x, \text{nil}), 1, x, x, x) \\
P((x, xs), n, sum, max, min) &= (\pi_1, P(xs, n', sum', max', min')) \\
&\quad (\pi_2, Ad(n', 1, n)) \\
&\quad (\pi_3, Ad(sum', x, sum)) \\
Q(l, av, max, min) &= (\pi_6, P(l, n, sum, max, min)) \\
&\quad (\pi_7, Div(sum, n, av))
\end{aligned}$$

Only a part of the third clause of the predicate P is kept, the program points π_4 and π_5 are dropped because they are not useful in computing av (they are needed to compute the values for max' and min').

Assuming a set of pairs (π_i, v_i) , where π_i is a program point and v_i a variable, a backward slicing analysis produces the slice computing for each point π_i the same values as the initial program for the variable v_i .

In our framework, a property is expressed by a function which takes at least an argument being the co-domain of the semantics function (a derivation tree of type PT) and the result of the property is an abstract domain. The slicing property takes an additional argument to represent the slicing criterion (of type $\text{PP} \rightarrow \mathcal{P}(\text{Pvar})$) and the type of the result is $\mathcal{P}(\text{PP})$ because slices are represented by sets of program points. The slicing criterion is represented in our approach by the mapping from program points to relevant variables. Because of the slicing property, we need extra information. We introduce a set of variables of interest according to a program point (this set represents the value of variable that must be preserved for computing the corresponding term). The initial value of the set is \emptyset . The property propagates this information of type $\mathcal{P}(\text{Pvar})$ and finally the type of the property is:

$$\alpha_{sl} : \text{PT} \times (\text{PP} \rightarrow \mathcal{P}(\text{Pvar})) \times \mathcal{P}(\text{Pvar}) \rightarrow \mathcal{P}(\text{PP}) \times \mathcal{P}(\text{Pvar})$$

The slicing property α_{sl} for the logic programming language is presented in Figure 4. The property takes as arguments a derivation tree PT , plus two additional parameters $RV \in \text{PP} \rightarrow \mathcal{P}(\text{Pvar})$ and $D \in \mathcal{P}(\text{Pvar})$. The second argument RV (for Relevant Variables) is the slicing criterion mentioned above. A program point π associated with a non-empty set $RV(\pi)$ is called an observation point. The third argument D of the property represents the set of variables whose

values must be preserved in the output context⁴ (normal form) of the term, *i.e.* the set of variables that must be preserved in the result R_i of the evaluation of the derivation tree PT_i . In the initial call, D is the empty set. The function α_{sl} is called recursively on the intermediate derivation trees (PT_i) of the natural semantics and sets of observation variables.

The result of the property is a pair (S, N) with $S \in \mathcal{P}(\text{PP})$ and $N \in \mathcal{P}(\text{Pvar})$. S is the set of program points of the term T that must be kept in the slice and N is the set of variables whose value must be preserved in the input context⁵. A program point must be kept in the slice if it can influence an observation point or the value of a variable of D in the output context. The same condition applies to decide which variables must be preserved in the input context. If the program point can be removed from the slice, the result of the property is (\emptyset, D) , which means that no program point is added to the slice and the variables whose values must be preserved in the input context are the variables that are necessary in the output context. Otherwise, the first component of the result of the property is $\bigcup_i S_i \cup \{\pi\}$ because π has to be added to the program points collected in the subterms of T . The second component N of the result is the set of variables whose value must be preserved in the input context C . It contains at least the set D and the variables $RV(\pi)$ of slicing criterion, thus we factorise that by setting $D' = D \cup RV(\pi)$ in beginning of the slicing definition.

We assume that the definitions of S_i and N_i are not mutually recursive. The definition of the sets of observation variables (third argument of α_{sl}) do not use N_j , $j > i$. Note that this is a characteristic feature of a backward analysis.

In Figure 4, the relation $\text{Indep}(C, D_1, D_2)$ is used to ensure that two sets of variables D_1 and D_2 are independent, which is the case when they do not share any renaming variables (in any substitution of the context C). The relation Indep appears in the first two cases as a necessary condition to exclude the term from the slice. If the relation holds, then the (renaming variable) substitution resulting from the evaluation of the term cannot have any impact on the variables of D . The relation $\text{UF}(C, x, t)$ is satisfied if the unification of x and t cannot fail for any substitution of C . It is a prerequisite for excluding $\text{Eq}(x, t)$ from the slice because a failure is recorded in the substitution tree as the \perp substitution⁶; as a consequence, it has an impact on all the variables. This condition was not included in the Op case, assuming that the logic programming language is equipped with mode annotations ensuring that operators are always called with their first two arguments ground and the last one free⁷. In both the Op and the Eq cases, the set of necessary variables (at the input of the program point) is D' added to all the program variables of the term: the set $\{x_1, x_2, x_3\}$ for

⁴ For a forward property this argument would characterise the input context rather than the output context.

⁵ For a forward property this argument would characterise the output context rather than the input context.

⁶ Note that \perp is an absorbing element for the semantics of the language. For instance $op(\perp, x_1, x_2, x_3) = \perp$ and $\text{unif}(\perp, x, t) = \perp$.

⁷ Otherwise an extra condition based on UF can be added as in the Eq case.

$\text{Op } (x_1, x_2, x_3)$ and the set of program variables occurring in t increased with x for the rule **Eq** (x, t) . The formal definitions of *Indep* and *UF* are presented in the bottom of Figure 4.

For the rule **And**, both branches are processed in turn (the second branch first since our property is computed in a backward direction). The property is first called with PT_2 and D' and the result is (S_2, N_2) ; then the property is computed with PT_1 and N_2 , we have (S_1, N_1) as the result. The program point π can be removed from the slice when both S_1 and S_2 are empty sets. When the program point is kept, the result of the operator **And** is then $(S_1 \cup S_2 \cup \{\pi\}, N_1)$ because the information about program points of both branches is kept and the set N_1 represents the variables must be preserved in the input context since we consider a backward direction.

The treatment of **Or** is different: the term is systematically kept in the slice because it always influences the values of all the variables (through the introduction of subtrees in the derivation tree). Both branches are computed independently and the result gathers the information of these two branches.

The rules for **Exists** and **Call** are not surprising. We assume that the variable x in **Exists** (x, U_1) is unique in a normalised program; so x can be removed from the set of necessary variables yielded by the analysis of U_1 (hence $N_1 - \{x\}$).

In the rule for **Call**, first the derivation tree corresponding to the predicate P_k is computed with the set $\{x_i \mid \neg \text{Indep}(C, D', \{y_i\})\}$ of variables to be preserved (i.e. the formal parameters x_i of P_k bounded to arguments y_i which are not independent from the set D'). The test in the rule for **Call** is similar to the test in the **Op** case. We could make more sophisticated choices to avoid including all the variables y_1, \dots, y_n in the set of the necessary variables.

4 Derivation of the Dynamic *on the Fly* Analyser

We have presented in section 2 the semantics function \mathcal{S} and the property α_{sl} in functional form in section 3. The general organisation is described by the following diagram:

$$\begin{array}{ccc} C \times T & \xrightarrow{\mathcal{S}} & PT \\ & \searrow \nu_a & \downarrow \alpha_{sl} \\ & & D_a \end{array}$$

The composition of the property α_{sl} and the semantics \mathcal{S} is a function of type $C \times T \rightarrow D_a$, where D_a is the domain of abstract values, the result of the analysis. This function computes successively the derivation tree related to a program, then the property of interest for this tree. It corresponds to a dynamic analysis *a posteriori* that inspects the trace produced after the program execution. It is interesting to formally describe dynamic analysers, because they are useful for instrumentation or debugging. We could also prefer dynamic analyses which, calculate their result *on the fly* i.e. during program execution. Their advantage is that they do not have to memorise traces before analysing them.

```

 $\alpha_{sl}(PT, RV, D) =$ 
let  $\pi = PT.stt.t.pp$ 
     $C = PT.stt.c$ 
     $D' = D \cup RV(\pi)$ 
in case  $(PT.lpt, PT.rn)$  of
  (nil, Op) : let Op  $(x_1, x_2, x_3) = PT.stt.t.i$ 
              in if  $RV(\pi) = \emptyset$  and  $Indep(C, D, \{x_3\})$ 
                  then  $(\emptyset, D)$ 
                  else  $(\{\pi\}, D' \cup \{x_1, x_2, x_3\})$ 
  (nil, Eq) : let Eq  $(x, t) = PT.stt.t.i$ 
              in if  $RV(\pi) = \emptyset$  and  $UF(C, x, t)$  and  $Indep(C, D, Pv(t) \cup \{x\})$ 
                  then  $(\emptyset, D)$ 
                  else  $(\{\pi\}, D' \cup Pv(t) \cup \{x\})$ 
   $([PT_1, PT_2], \wedge)$  : let  $(S_2, N_2) = \alpha_{sl}(PT_2, RV, D')$ 
                         $(S_1, N_1) = \alpha_{sl}(PT_1, RV, N_2)$ 
                        in if  $RV(\pi) = \emptyset$  and  $S_1 \cup S_2 = \emptyset$ 
                            then  $(\emptyset, D)$ 
                            else  $(S_1 \cup S_2 \cup \{\pi\}, N_1)$ 
   $([PT_1, PT_2], \vee)$  : let  $(S_2, N_2) = \alpha_{sl}(PT_2, RV, D')$ 
                         $(S_1, N_1) = \alpha_{sl}(PT_1, RV, D')$ 
                        in  $(S_1 \cup S_2 \cup \{\pi\}, N_1 \cup N_2)$ 
   $(PT_1, \exists)$  : let Exists  $(x, U_1) = PT.stt.t.i$ 
                 $(S_1, N_1) = \alpha_{sl}(PT_1, RV, D')$ 
                in if  $RV(\pi) = \emptyset$  and  $S_1 = \emptyset$ 
                    then  $(\emptyset, D)$ 
                    else  $(S_1 \cup \{\pi\}, N_1 - \{x\})$ 
   $(PT_1, Call)$  : let Call  $(P_k(y_1, \dots, y_n)) = PT.stt.t.i$ 
                   $(S_1, N_1) = \alpha_{sl}(PT_1, RV, \{x_i \mid \neg Indep(C, D', \{y_i\})\})$ 
                  in if  $RV(\pi) = \emptyset$  and  $S_1 \cup N_1 = \emptyset$  and  $Indep(C, D, \{y_1, \dots, y_n\})$ 
                      then  $(\emptyset, D)$ 
                      else  $(S_1 \cup \{\pi\}, D' \cup \{y_1, \dots, y_n\})$ 

 $UF(C, x, t) = \forall \theta \in C. \theta \neq \perp \Rightarrow \exists \sigma = mgu(\theta(x), \theta(t))$ 
 $Pv(t) =$  set of program variables occurring in  $t$ 
 $Rv(rt) =$  set of renaming variables occurring in  $rt$ 
 $Indep(C, D_1, D_2) = \forall \theta \in C. \theta \neq \perp \Rightarrow \{Rv(\theta(x)) \mid x \in D_1\} \cap \{Rv(\theta(x)) \mid x \in D_2\} = \emptyset$ 

```

Fig. 4. Slicing property

The derivation of a dynamic *on the fly* analyser from a dynamic analyser *a posteriori* presents similarities with a well-known program transformation within the framework of functional programming. The program transformation is called deforestation [28] and its purpose is to eliminate the intermediate data structures induced by the composition of recursive functions. Here, the intermediate structure is the derivation tree of the natural semantics. We use folding and unfolding transformations to carry out deforestation. The three principal operations are the following:

- unfoldings: we set $\nu_a(C, T) = \alpha_{sl}(\mathcal{S}(C, T))$ and replace in the expression the calls to the recursive functions α_{sl} and \mathcal{S} by their definition.
- applications of laws on the operators of the language (like the conditional ones, the expressions `case` and `let`).
- foldings which consist in replacing the occurrences of $\alpha_{sl}(\mathcal{S}(C', T'))$ from calls to $\nu_a(C', T')$.

The goal of these transformations is to remove all the calls to the property extraction function α_{sl} , to obtain a closed definition of $\nu_a(C, T)$. The function obtained is then a dynamic *on the fly* analyser since it does not build the intermediate derivation trees any more.

The partial correction of the transformation by folding/unfolding is obvious. The total correction is not assured in general because some inopportune foldings can introduce cases of non-termination. The Improvement Theorem in [20] can be extended to a method (*the extended improved unfold-fold method*) presented in [21] which makes it possible to show the total correction of the method proposed in this paper.

Dynamic Slicing Analyser

The definition of the dynamic slicing analyser for the logic programming language is the following:

$$\mathcal{SL}_d(C, T, RV, D) = \alpha_{sl}(\mathcal{S}(C, T), RV, D)$$

First, we use an unfolding technique applied to the semantics and the property functions. We present in [11] the transformation rules used for the derivation of the dynamic *on the fly* analyser by unfolding. Figure 5 presents these unfoldings for two rules (the other cases are straightforward).

To obtain a dynamic *on the fly* analyser, we must apply folding steps that allows us to remove the calls of the function α_{sl} . Figure 6 presents the result of these foldings. The fact that \mathcal{SL}_d itself calls \mathcal{S} shows that it is a dynamic analysis.

```

 $\mathcal{SL}_d(C, T, RV, D) =$ 
let  $D' = D \cup RV(T.pp)$ 
in case  $T$  of
   $(\pi, \text{Op}(x_1, x_2, x_3))$  : if  $RV(\pi) = \emptyset$  and  $\text{Indep}(C, D, \{x_3\})$ 
    then  $(\emptyset, D)$ 
    else  $(\{\pi\}, D' \cup \{x_1, x_2, x_3\})$ 
   $(\pi, \text{And}(U_1, U_2))$  : let  $PT_1 = \mathcal{S}(C, U_1)$ 
     $R_1 = PT_1.\text{stt.nf}$ 
     $(S_2, N_2) = \alpha_{sl}(\mathcal{S}(R_1, U_2), RV, D')$ 
     $(S_1, N_1) = \alpha_{sl}(\mathcal{S}(C, U_1), RV, N_2)$ 
    in if  $RV(\pi) = \emptyset$  and  $S_1 \cup S_2 = \emptyset$ 
      then  $(\emptyset, D)$ 
      else  $(S_1 \cup S_2 \cup \{\pi\}, N_1)$ 

```

Fig. 5. Unfoldings of semantics and property functions

```

 $\mathcal{SL}_d(C, T, RV, D) =$ 
let  $D' = D \cup RV(T.pp)$ 
in case  $T$  of
   $(\pi, \text{Op}(x_1, x_2, x_3))$  : if  $RV(\pi) = \emptyset$  and  $\text{Indep}(C, D, \{x_3\})$ 
    then  $(\emptyset, D)$ 
    else  $(\{\pi\}, D' \cup \{x_1, x_2, x_3\})$ 
   $(\pi, \text{And}(U_1, U_2))$  : let  $PT_1 = \mathcal{S}(C, U_1)$ 
     $R_1 = PT_1.\text{stt.nf}$ 
     $(S_2, N_2) = \mathcal{SL}_d(R_1, U_2, RV, D')$ 
     $(S_1, N_1) = \mathcal{SL}_d(C, U_1, RV, N_2)$ 
    in if  $RV(\pi) = \emptyset$  and  $S_1 \cup S_2 = \emptyset$ 
      then  $(\emptyset, D)$ 
      else  $(S_1 \cup S_2 \cup \{\pi\}, N_1)$ 

```

Fig. 6. Dynamic (*on the fly*) slicing analysis

5 Related Work

The fold/unfold transformation framework used here is based on seminal work by Burstall and Darlington [26]. The application of the technique to the derivation of programs has also been investigated in [5], which presents the synthesis of several sorting algorithms. The initial specification is expressed in terms of sets and predicate logic constructs. Our transformations are also reminiscent of the deforestation technique [3,9,28]: in both cases the goal is to transform a composition of recursive functions into a single recursive definition.

Generic frameworks for program analysis have been proposed in the context of logic programming languages [15] and data flow analysis [26,30]. They rely on abstract interpretations of denotational semantics [15,26] or interpreters [30] and genericity is achieved by parameterising the abstract domains and choosing appropriate abstract functions. The implementation details of the analysis algorithm can be factorised. While these tools may attain a higher degree of mechanisation than our framework, they do not offer to the user the same level of abstraction: they take as input the *specification of an abstract interpreter* rather than the *specification of a property*. Despite this difference of point of view, all these works are obviously inspired by the same goals. The framework introduced in [24] is closer to the spirit of the work presented in this paper but the technique itself is quite different. Programs are represented as models in a modal logic and a data flow analysis can be specified as a property in the logic. An efficient data flow analyser can be generated by partially evaluating a specific model checker with respect to the specifying modal formula. In comparison with this work, our framework trades mechanisation against generality: it is not limited to data flow analyses but the derivation process by fold/unfold transformations is not fully automatic.

Few papers have been devoted to the semantics of program slicing so far. A relationship between the behaviour of the original program and the behaviour of the slice is proved in [19]. The semantics of the language is expressed in terms of program dependence graphs; thus the programs are first analysed in order to extract their dependences. This approach is well suited to the treatment of imperative languages. Formal definitions and a classification of different notions of slicing are provided in [27]. The main distinctions are backward vs forward analysers, executable vs non executable slices, and dynamic vs static analysers. Their definitions are based on denotational semantics and they focus on the specifications of the analyses. In [8] a description of a family of slicing algorithms generalising the notions of dynamic and static slice to that of a constrained slice is presented. Genericity with respect to the programming language is achieved through a translation into an intermediate representation called PIM. Programs are represented as directed acyclic graphs whose semantics is defined in terms of rewriting rules. Slicing is carried out using term graph rewriting with a technique for tracing dynamic dependence relations. It should be noted that a richer notion of slicing has been proposed for logic programming languages, which returns not only the set of program points that must be kept in the slice, but also the necessary variables at each program point [23]. This increased precision can also

be expressed in our framework, but we preferred to present the simpler version here for the sake of size and readability. By collecting the following information

$$(\bigcup_i S_i + \{(\pi, D \cup N)\}, N)$$

we can modify straightforwardly each rule in order to get the same precision as [23].

6 Conclusion

We have presented a method to derive dynamic analysers by program transformation (folding/unfolding). A dynamic analyser is expressed as composition of a semantics and a property functions. The analyser is called *a posteriori*, it is a function computing first a complete program execution trace (derivation tree) and then extracting the property of interest. A recursive definition of an analyser can be obtained by program transformation. This function is a dynamic *on the fly* analyser that computes the property during program execution.

We have focussed on dynamic analysis in the body of paper. Our generic dynamic analyser is defined in a strongly typed functional language [8]. As a consequence, we can rely on previous results on logical relations and abstract interpretation [14] in order to systematically construct static analysers from the dynamic analysers. The first task is to provide abstract domains for the static slicing analyser and the corresponding abstraction functions. We recall that the type of the dynamic analyser is $\mathcal{C} \times \mathcal{T} \times (\mathcal{PP} \rightarrow \mathcal{P}(\mathcal{Pvar})) \times \mathcal{P}(\mathcal{Pvar}) \rightarrow \mathcal{P}(\mathcal{PP}) \times \mathcal{P}(\mathcal{Pvar})$. Since $\mathcal{PP} \rightarrow \mathcal{P}(\mathcal{Pvar})$, $\mathcal{P}(\mathcal{Pvar})$ and $\mathcal{P}(\mathcal{PP})$ are already abstract domains associated with the dynamic analysis, only \mathcal{C} needs to be abstracted [9]. The next stage to derive a correct static analyser is to find appropriate abstractions for the constants and operators occurring in the definition of the analyser. It is shown in [1] that the correctness of the abstract interpretation of the constants and operators of the language entails the correctness of the abstract interpretation of the whole language. The correctness of the abstract interpretation means that the results of the dynamic analysis and the static analysis are related if their arguments are. In fact, it is possible to define the most precise abstraction for each constant and operator of the language [1]. The basic idea to find the best abstraction $op^a(v_1^a, \dots, v_n^a)$ of an operator op is to define it as the least upper bound of the abstractions of all the results of op applied to arguments v_i belonging to the concretisation sets of the arguments of the v_i^a . The technique sketched here provides a systematic way to construct a correct abstract interpretation, and thus to derive a static analyser from a dynamic analyser [10, 12]. By deriving static analysers as abstractions of dynamic analysers, we can see the dynamic

⁸ Note that the typing mentioned here has nothing to do with the language in which the analysed programs are written, this language itself can perfectly well be untyped.

⁹ Of course, as usual in abstract interpretation, $\mathcal{PP} \rightarrow \mathcal{P}(\mathcal{Pvar})$, $\mathcal{P}(\mathcal{Pvar})$ and $\mathcal{P}(\mathcal{PP})$ can also be abstracted if further approximations are needed, but we do not consider this issue here.

analyser either as an intermediate stage in the derivation of a static analyser (playing a role similar to a collecting semantics) or as the final product of the derivation.

The theory of abstract interpretation [4] provides a strong formal basis for static program analysis. The work described here does not provide an alternative formal underpinning for program analysis. Its goal is rather to put forward a derivation approach for the design of analysers from high level specifications.

Our framework is applicable to a wide variety of languages, properties and *type of service* (dynamic or static). We have proposed in the body of the paper a formal definition of a dynamic slicing analyser for a logic programming language. To our knowledge, this definition is the first one to be formal, so the benefit of our approach is striking in this case. In [10], we present the derivation of dynamic and static analysers for a strictness analysis of a higher-order functional language and a live variable analysis for an imperative language. We have also applied this work for a globalisation analysis of a higher-order functional language and a generic sharing analysis. Pushing our approach ever further we arrive at a natural semantics format and a format for slicing, as presented in [12]. We have shown the correctness of the slicing property format. These formats can be instantiated for several programming languages (imperative language, logic programming language and functional language). The slicing property for the logic programming that we have presented here is an instantiation of the slicing format.

As mentioned in the introduction, we wanted to establish the connection between the result of the analysis and its intended use. Analyses are generally performed to check assumptions about the behaviour of the program at specific points of its execution or to enable program optimisations. In both cases the intention of the analysis can be expressed in terms of a transformation and a relation as presented in [10,12]. The transformation depends on the result of the analysis and the relation establishes a correspondence between the semantics of the original program and the transformed program. For example, in the case of a program analysis for compiler optimisation the transformation expresses the optimisation that is allowed by the information provided by the analysis and the relation is the equality between the final results (or outputs) of the original and the transformed program. It is not always the case that the relation is the equality: a counter-example is slicing analysis described in this paper (because the new program is required to behave like the original one only with respect to specific program points and variables). We have formally defined and proved in [12] a property for the intention of a slicing analysis but space considerations prevent us from presenting the intentional property for slicing.

There is a main aspect in which the work described here may seem limited: we have used only natural semantics and terminating programs. Structural Operational Semantics (SOS) are more precise than natural semantics and they are required for a proper treatment of non-determinism, non-termination and parallelism [17]. In fact, the natural semantics introduced in section 2 can be

replaced by SOS without difficulty¹⁰ and the dynamic analyses can be defined in the very same way. The extra difficulty introduced by SOS is the fact that they create new program fragments which makes it necessary to abstract over the syntax of the language to derive a static analyser. This problem is discussed in [22]. We can also adapt our natural semantics to SOS by using the technique presented in [13]. To achieve this goal, the classical inductive interpretation of natural semantics has to be extended with coinduction mechanisms and rules must be defined to express divergence.

Acknowledgements

I would like to thank Pascal Fradet, Thomas Jensen, Daniel Le Métayer and Ronan Gaugne for their useful comments. Thanks are also due to the anonymous referees SAS for their criticisms on an earlier draft.

References

1. S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1:5–40, 1990. [130] [130] [130]
2. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977. [129]
3. W.N. Chin. *Automatic methods for program transformation*. PhD thesis, Imperial College, 1990. [129]
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977. [130] [131]
5. J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978. [129]
6. J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976. [129]
7. T. Despeyroux. Typol: a formalism to implement natural semantics. Technical Report 94, INRIA, France, 1988. [115]
8. J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *POPL*, pages 379–392, 1995. [129]
9. A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *FPCA*, pages 223–232, 1993. [129]
10. V. Gouranton. *Dérivation d’analyseurs dynamiques et statiques à partir de spécifications opérationnelles*. PhD thesis, Université de Rennes, France, 1997. [116] [122] [130] [131] [131]
11. V. Gouranton. Deriving analysers by folding/unfolding of natural semantics and a case study: slicing. Technical Report 3413, INRIA, France, 1998. [127]
12. V. Gouranton and D. Le Métayer. Dynamic slicing: a generic analysis based on a natural semantics format. Technical Report 3375, INRIA, France, 1998. [115] [130] [131] [131] [131]

¹⁰ In order to deal with SOS, we basically need to change the type `NF` in `STT` and to introduce a global loop in the semantics since a SOS rule represents a single evaluation step.

13. H. Ibraheem and D. A. Schmidt. Adapting big-step semantics to small-step style: coinductive interpretations and "higher-order" derivations. In *Second Workshop on Higher-Order Techniques in Operational Semantics (HOOTS2)*, 1997. [132](#)
14. G. Kahn. Natural semantics. In *STACS 87*, number 247 in Lecture Notes in Computer Science, pages 22–39. Springer-Verlag, 1987. [115](#)
15. B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A generic interpretation algorithm and its complexity analysis. In *ICLP*, pages 64–78, 1991. [118](#), [129](#), [129](#)
16. D. Miller and G. Nadathur. Higher-order logic programming. In *ILPC*, volume 225 of *LNCS*, pages 448–462, 1986. [117](#)
17. H. Riis Nielson and F. Nielson. *Semantics With Applications*. John Wiley & Sons, 1992. [131](#)
18. T. Reps and T. Turnidge. Program specialization via program slicing. In *International Seminar on Partial Evaluation*, 1996. [122](#)
19. T. Reps and W. Yang. The semantics of program slicing. Technical Report 777, University of Wisconsin, Madison, 1988. [129](#)
20. D. Sands. Proving the correctness of recursion-based automatic program transformations. In *TAPSOFT*. Springer-Verlag, 1995. [127](#)
21. D. Sands. Total correctness by local improvement in the transformation of functional programs. *TOPLAS*, 18:175–234, 1996. [127](#)
22. D.A. Schmidt. Abstract interpretation of small-step semantics. In *5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*. LNCS, 1996. [132](#)
23. S. Schoenig and M. Ducassé. A backward slicing algorithm for prolog. In *SAS*, number 1145 in LNCS, pages 317–331. Springer-Verlag, 1996. [122](#), [129](#), [130](#)
24. B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Science*, 21(2):115–139, 1993. [129](#)
25. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995. [122](#)
26. G.A. Venkatesh. A framework for construction and evaluation of high-level specifications for program analysis techniques. In *PLDI*, volume 24, pages 1–12, 1989. [129](#), [129](#)
27. G.A. Venkatesh. The semantics approach to program slicing. In *PLDI*, pages 107–119, 1991. [129](#)
28. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990. [127](#), [129](#)
29. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 4:352–357, 1984. [122](#)
30. K. Yi and W. L. Harrison III. Automatic generation and management of interprocedural program analyses. In *POPL*, pages 246–259, 1993. [129](#), [129](#)

A Symbolic Semantics for Abstract Model Checking^{*}

Francesca Levi

Dipartimento di Informatica, Università di Pisa
and

LIX, École Polytechnique
levifran@di.unipi.it

Abstract. We present a finite symbolic semantics of value-passing concurrent processes, that can be suitably interpreted over abstract values to compute a lower approximate semantics of full μ -calculus. The main feature of the semantics is that classical branching is replaced by explicit relations of non-deterministic and alternative choices among transitions. A combination of safe upper and lower approximations of the basic operators of the logic is used to handle negation. The relations of non-deterministic and alternative choices turn out to be very useful for the dual approximations of the existential next modality.

Key words: Model checking, μ -calculus, abstract interpretation.

1 Introduction

Model Checking is a very successful technique for the automatic verification of temporal properties of reactive and concurrent systems, but it is only applicable to finite-state systems. Over the past few years, abstract interpretation has been widely applied to handle large as well as infinite systems with model checking [3, 19, 11, 13, 4, 8, 17, 10, 15]. Abstract interpretation [6, 7] was originally conceived in the framework of data-flow analysis for designing approximate semantics of programs and relies on the idea of obtaining an approximate semantics from the standard one by substituting the concrete domain of computation and its basic operations with an abstract domain and corresponding abstract operations. The typical approach consists of constructing an *abstract model* over a chosen set of abstract states that can be used in model checking instead of the concrete one. To this aim the abstract model has to be safe, namely the formulas satisfied by the abstract model have to hold in the concrete one. For branching time logics the definition of a safe abstract transition relation among abstract states presents some basic difficulties and a single abstract transition relation cannot preserve both the existential and the universal next modality. Several authors [11, 10, 4] propose to adopt two different abstract transition relations: a

^{*} This work has been partially supported by the HCM project ABILE (ER-BCHRXCT940624).

free transition relation for computing the universal next modality, and a *constrained* transition relation for computing the existential next modality. Safeness of the free transition relation is ensured, if every concrete transition induces a free transition among the corresponding abstract states. In contrast, a constrained transition between abstract states is safe only if all the corresponding concrete transitions exist. Given this notion of safeness it turns out to be very difficult to effectively compute a sufficiently precise safe abstract model without constructing the concrete one.

In this paper we propose a method for applying abstract interpretation to the μ -calculus model checking of value-passing concurrent processes. The main contribution is the definition of a symbolic semantics of processes in the style of [12], whose main feature is that explicit relations of non-determinism and alternative choice among transitions replace classical branching. Moreover, a finite graph for regular processes is achieved by avoiding the infinite paths of [12] due to parameterized recursion. Model checking of μ -calculus can be suitably performed by interpreting the obtained symbolic graph over concrete environments assigning concrete values to variables. However, since processes are capable of exchanging values taken from a typically infinite set, the fixpoint computation of μ -calculus semantics is not effectively computable. We define a technique to compute a lower approximation of the μ -calculus semantics by interpreting the symbolic graph over abstract environments on a given (finite) set of abstract values. Safeness of the lower approximation ensures indeed the preservation of any property. Following [13] the lower approximation is achieved by combining dual safe upper and lower abstract functions corresponding to all logical connectives except negation. The critical case is undoubtedly that of the next modality, where safe constrained and free transition relations among abstract processes have to be considered. We show that explicit non-deterministic and alternative choices between transitions allow us to avoid some typical problems due to abstract branching so that a more precise lower approximation of the next modality in particular is achieved with respect to previous proposals [10]. Finally, we discuss the basic problems that typically lead to miss optimality in the approximations of the next modalities.

The paper is organized as follows. Section 2 presents value-passing concurrent processes, μ -calculus and concrete model checking. Section 3 summarizes the basic concepts of abstract interpretation. The symbolic graph is described in Sect. 4 and the corresponding model checking algorithm is shown in Sect. 5. Section 6 presents abstract model checking and Sect. 7 discuss optimality of abstract model checking.

2 Concrete Model Checking

We consider a value-passing version of CCS. Let *Val* a set of values (possibly infinite), *Chan* a set of channels and *Var* a set of variables. Moreover, let *Bexp* and *Vexp* be sets of boolean and values expressions. Processes *Proc* are generated

by the following grammar

$$p ::= nil \mid x \mid a.p \mid be \nabla p_1, p_2 \mid p_1 \times p_2 \mid p_1 + p_2 \mid p \setminus L \mid P(e_1, \dots, e_n)$$

where $a \in \{cle, c?v, \tau \mid c \in Chan, e \in Vexp, v \in Var\}$, $L \subseteq Chan$, and $be \in Bexp$ is a boolean expression and $e_i \in Vexp$ are expressions over values. Process $c?v.p$ will receive a value v on the channel c and then behaves as $p[v/x]$, where $p[v/x]$ denotes the standard substitution of v for all free occurrences of x . Process $cle.p$ will send the value of the expression e and then behaves as p . The operator $+$ represents choice, while \times represents parallel composition. Process $be \nabla p_1, p_2$ behaves as p_1 if the value of be is true, and as p_2 otherwise. The operator $\setminus L$ is the standard restriction for a set of channels L . Finally, $P(x_1, \dots, x_n)$ is a process constant, which has an associated definition $P(x_1, \dots, x_n) \equiv p$. We assume the usual definitions of free variables $fv(p)$ and bound variables $bv(p)$ of processes. A process p is closed iff $fv(p) = \emptyset$. In the following, we denote by capital letters T, P, \dots open processes. For recursive processes $P(x_1, \dots, x_n) \equiv p$ we assume $fv(p) \subseteq \{x_1, \dots, x_n\}$ and we assume the body to be guarded. The concrete semantics of processes is defined in Table 1 of the appendix as a labelled transition system $LTS(p) = (P^*, \xrightarrow{a})$ with actions $a \in Act = \{\tau, c?v, c!v \mid c \in Chan, v \in Val\}$. Two semantic functions $\mathcal{S}_v : Vexp \rightarrow Val$ and $\mathcal{S}_b : Bexp \rightarrow \{tt, ff\}$ for the evaluation of expressions are used. For $a \in Act$, we define $chan(\tau) = \emptyset$, $chan(c?v) = chan(c!v) = \{c\}$. Moreover, we denote by \bar{a} the symmetric action of a , namely $c!v$ for $c?v$ and $c?v$ for $c!v$. Note that, if the set of values is infinite, the labelled transition system is infinite and infinitely branching.

For expressing temporal properties of processes we consider a simple extension of propositional μ -calculus [14]. Let Act be a set of actions and VAR be a set of logical variables. Formulas are inductively defined as follows.

$$A ::= X \mid A \wedge A \mid < K > A \mid \neg A \mid \mu X.A$$

where $X \in VAR$ is a logical variable and $K \in \{\tau, c?v, c!v \mid c \in Chan, V \subseteq Val\}$. We assume that each variable X occurs positively in formulas. The modality $< K >$ subsumes the classical existential next modality $< a >$ ranging over actions $a \in Act$ and corresponds to $\bigvee_{a \in K} < a >$. The dual universal modality is $[K] \equiv \neg < K > \neg$. The operator $\mu X.A$ denotes the least fixpoint and the dual operator of greatest fixpoint is equivalent to $\nu X.A \equiv \neg \mu X.A[\neg X/X]$. Note that formulas with K infinite are needed for subsuming the logics CTL and CTL^* , since Act can be infinite. For instance, the classical liveness property $\forall F A$ can be expressed as $\mu X.A \vee [\tau] X \bigvee_{c \in Chan} [c?v] X \bigvee_{c \in Chan} [c!sVal] X$.

Traditional global model checking corresponds to compute the semantics of the formula $\| A \|$ on the concrete labelled transition system $LTS(p) = (P^*, \xrightarrow{a})$. Let $\delta : VAR \rightarrow \mathcal{P}(P^*)$ be a valuation assigning subsets of P^* to logical variables. The semantics of an open formula A with respect to δ is defined as:

$$\begin{aligned} \| X \|_{\delta} &= \delta(X) & \| \neg A \|_{\delta} &= P \setminus \| A \|_{\delta} \\ \| A_0 \wedge A_1 \|_{\delta} &= \| A_0 \|_{\delta} \cap \| A_1 \|_{\delta} & \| \mu X.A \|_{\delta} &= \mu V. (\| A \|_{\delta[V/X]}) \\ \| < K > A \|_{\delta} &= \bigcup_{a \in K} \| < a > \| (\| A \|_{\delta}) \end{aligned}$$

where $\delta[V/X]$ stands for the valuation δ' , which agrees with δ except that $\delta'(V) = \delta(X)$. The *next modality function* $\|<a>\|: \mathcal{P}(P^*) \rightarrow \mathcal{P}(P^*)$ is given by $\|<a>\|(S) = \{p \in P \mid \exists p' \xrightarrow{a} p', p' \in S\}$.

Therefore, $p \in \|A\| (p \models A)$ iff $p \in \|A\|_{\delta_0}$, where δ_0 is the empty evaluation.

3 Abstract Interpretation Theory

In this section we briefly recall the basic ideas of abstract interpretation, we refer the reader to [6,7] for more details. The theory of abstract interpretation provides a systematic method to design approximate semantics of programs by replacing the concrete domain of computation with a simpler abstract domain. The relation between the concrete and the abstract domain is precisely stated into a formal framework.

Let (C, \leq) and $(A, \leq^\#)$ be two posets, where orderings \leq and $\leq^\#$ correspond to precision. A pair of functions (α, γ) , where $\alpha: C \rightarrow A$ (abstraction) and $\gamma: A \rightarrow C$ (concretization) is called a *Galois connection* iff $\forall c \in C, \forall a \in A, \alpha(c) \leq^\# a \Leftrightarrow c \leq \gamma(a)$. These requirements can also be captured by saying that α is extensive ($c \leq \gamma(\alpha(c))$), γ is reductive ($\alpha(\gamma(a)) \leq^\# a$), α and γ are total and monotonic. If $\alpha(\gamma(a)) = a$, then (α, γ) is called a Galois insertion.

Intuitively, the condition $c \leq \gamma(\alpha(c))$ ensures the loss of information of abstraction to be safe. On the other hand, condition $\alpha(\gamma(a)) \leq^\# a$ ensures that the concretization process introduce no loss of information. Let $\mathcal{S}(P)$ be the semantics of a program P computed as the least fixpoint of a semantic function F over the concrete domain (C, \leq) . The goal is that of computing an approximate semantics $\mathcal{S}^\#(P)$ over $(A, \leq^\#)$, that is *safe* $\alpha(\mathcal{S}(P)) \leq^\# \mathcal{S}^\#(P)$. The main result is that a safe approximate semantics $\mathcal{S}^\#(P)$ can be computed as the least fixpoint of a safe approximate semantic function $F^\#$ over $(A, \leq^\#)$, such that $\alpha(F(c)) \leq^\# F^\#(\alpha(c))$, for each $c \in C$. Moreover, it has been shown there there exists always a best approximate semantic function $F^\#$ (*optimal*), when $\alpha(F(c)) = F^\#(\alpha(c))$. In the Galois insertion case this is equivalent to $\alpha(F(\gamma(a))) = F^\#(a)$, for each $a \in A$.

4 The Symbolic Graph

In this section we define the symbolic graph of processes. Symbolic semantics as introduced in [12] relies on the idea of using *symbolic actions* instead of concrete actions. For instance, transitions modeling input are represented by a single transition $c?x.p \xrightarrow{c?x} p$. To handle conditional open processes symbolic semantics depend in addition on boolean guards. Thus, a symbolic transition $T \xrightarrow{(c,\theta)} T'$ represents all concrete transitions with action a corresponding to symbolic action θ for the assignments to the free variables of T such that guard c is satisfied. Our definition of the symbolic graph differs in some aspects from the classical one. First, classical branching is replaced by explicit relations of non-deterministic (\oplus) and alternative choices (\otimes) among transitions. Moreover, a method to avoid

infinite applications of the standard rule for recursion is proposed. This way a finitely branching and finite graph for regular processes is obtained.

We use the standard notions of substitutions and environments. A *substitution* is a partial function $\sigma : Var \rightarrow VExp$ and a *simple substitution* is an injective $\sigma : Var \rightarrow Var$. We denote by *Sub* the set of simple substitutions. For a substitution σ we denote by $tar(\sigma)$ and $dom(\sigma)$ its target and source, respectively. An *environment* is a total function $\rho : Var \rightarrow Val$ and *Env* is the set of environments. For an environment ρ we denote by $\rho[x \rightarrow v]$ the environment that agrees with ρ except that the value assigned to variable x is v . Let T be an open process and $x \in Var$. We say that x is *fresh* in T iff $x \notin fv(T) \cup bv(T)$. Moreover, we say that a term T is free for a simple substitution σ with $dom(\sigma) \subseteq fv(T)$, iff for each $x \in fv(tar(\sigma))$, $x \notin bv(T) \cup (fv(T) \setminus tar(\sigma))$. Let T be an open process and ρ an environment. We denote by (T, ρ) the closed process $T\rho$, that is obtained by substituting $\rho(x)$ to variable x for each $x \in fv(T)$.

Let \mathcal{C} be the set of *constraints* with $c ::= be \mid e = e \mid e \in V \mid true \mid \neg c \mid c \wedge c$, where $e \in Vexp$, $be \in Bexp$ and $V \subseteq Val$. We denote by $vars(c)$ the variables occurring in constraint c . For $c \in \mathcal{C}$ we consider the semantic function $\| c \| : \mathcal{P}(Env) \rightarrow \mathcal{P}(Env)$ obtained in the trivial way by $\| be \| = \{\rho \in Env \mid \mathcal{S}_b(be\rho) = tt\}$, $\| e_1 = e_2 \| = \{\rho \in Env \mid \mathcal{S}_v(e_1\rho) = \mathcal{S}_v(e_2\rho)\}$ and $\| e \in V \| = \{\rho \in Env \mid \mathcal{S}_v(e\rho) \in V\}$. We use the notation $\rho \models c$ for $\rho \in \| c \|$.

Let the *symbolic actions* be $SymAct = \{c?x, c!e, \tau \mid c \in Chan, x \in Var, e \in Vexp\}$ with $bv(c?x) = \{x\}$, $bv(c!e) = bv(\tau) = \emptyset$ and $fv(c!e) = vars(e)$, $fv(\tau) = fv(c?x) = \emptyset$.

Transitions of the symbolic semantics are $T \xrightarrow{\otimes_{i \in \{1, n\}} \Theta_i} \otimes_{i \in \{1, n\}} \Omega_i$, such that for each $i \in \{1, n\}$ ($i \in \{1, \dots, n\}$)

1. $\Theta_i = (c_i, \oplus_{j_i \in \{1, n_i\}} \theta_{i, j_i})$ with $\theta_{i, j_i} \in SymAct \cup \{*\}$ and $c_i \in \mathcal{C}$;
2. $\Omega_i = \oplus_{j_i \in \{1, n_i\}} T_{i, j_i}$ where T_{i, j_i} are open processes;
3. $vars(c_i) \subseteq fv(T)$, $fv(\theta_{i, j_i}) \subseteq fv(T)$ and $fv(T_{i, j_i}) \subseteq fv(T) \cup bv(\theta_{i, j_i})$.

All possible behaviors of closed processes obtained from T are represented by a single transition $T \xrightarrow{\otimes_{i \in \{1, n\}} \Theta_i} \otimes_{i \in \{1, n\}} \Omega_i$, where alternative choices are related by \otimes and non-deterministic choices by \oplus . The idea is that for each environment ρ there exists a unique alternative c_i that is satisfied and symbolic actions θ_{i, j_i} with corresponding processes T_{i, j_i} represent the concrete transitions of (T, ρ) .

Let us explain informally the construction of transitions. The complete semantic rules are shown in Table 2 of the appendix.

Transitions of a basic process $a.T$ are obtained by a rule $a.T \xrightarrow{(true, a \oplus *)} T \oplus a.T$, where the special action $*$ denotes idle action and is used in the parallel composition rule.

The non-deterministic choice of $+$ is reflected by the composition of transitions with \oplus . The rule for choice is as follows

$$\frac{\{T_i \xrightarrow{\otimes_{j_i \in \{1, n_i\}} \Theta_{i, j_i}} \otimes_{j_i \in \{1, n_i\}} \Omega_{i, j_i}\}_{i \in \{1, 2\}}}{T_1 + T_2 \xrightarrow{\otimes_{i \in \{1, 2\}} \otimes_{j_i \in \{1, n_i\}} \Theta_{j_1, j_2}^+} \otimes_{i \in \{1, 2\}, j \in \{1, n_i\}} \Omega_{j_1, j_2}^+} +$$

where Θ_{j_1, j_2}^+ is guarded by constraint $c_{1, j_1} \wedge c_{2, j_2}$ and where all non-deterministic choices of $\Theta_{i, j_i} = (c_{i, j_i}, \oplus_{h_i \in \{1, k_{j_i}\}} \theta_{i, j_i, h_i})$ for $i \in \{1, 2\}$ are merged by \oplus . The resulting processes are analogously combined by Ω_{j_1, j_2}^+ . In fact, for each environment that satisfies both guards both actions of T_1 and of T_2 can be chosen.

Example 1. For instance, a process $T = c!x.T_1 + a!x.T_2$ is modeled by $c!x.T_1 + a!x.T_2 \xrightarrow{(true, c!x \oplus a!x \oplus *)} T_1 \oplus T_2 \oplus T$.

The alternative choices of a conditional process are related by \otimes through the following rule

$$\frac{\{T_i \xrightarrow{\otimes_{j_i \in \{1, 2\}} \Theta_{i, j_i}} \otimes_{j_i \in \{1, n_i\}} \Omega_{i, j_i}\}_{i \in \{1, 2\}}}{be \nabla T_1, T_2 \xrightarrow{\otimes_{i \in \{1, 2\}} \bar{\Theta}_{i, j_i}} \otimes_{i \in \{1, 2\}, j_i \in \{1, n_i\}} \bar{\Omega}_{i, j_i}} \nabla$$

where $\bar{\Theta}_{1, j_1}$ is equivalent to Θ_{1, j_1} where the guard is additionally constrained by be , while $\bar{\Theta}_{2, j_2}$ is equivalent to Θ_{2, j_2} where the guard is additionally constrained by $\neg be$. Intuitively, transitions of $be \nabla T_1, T_2$ are either transitions of T_1 , if be is satisfied, or transitions of T_2 , if be is not satisfied.

Example 2. For $T = x > 0 \nabla c!x.T_1 + a!x.T_2, b!x.nil$ we have $T \xrightarrow{\Theta_1 \otimes \Theta_2} \Omega_1 \otimes \Omega_2$, where $\Theta_1 = (x > 0, c!x \oplus a!x \oplus *)$, $\Theta_2 = (x \leq 0, b!x \oplus *)$, $\Omega_1 = T_1 \oplus T_2 \oplus T$ and $\Omega_2 = nil \oplus T$. In fact, for each environment ρ either $x > 0$ or $x \leq 0$ is true and the process (T, ρ) is able to perform either both $c!\rho(x)$ and $a!\rho(x)$ or $b!\rho(x)$, respectively.

The rule of parallel composition is quite complex. Since a single transition must represent all behaviors combined by the relations \oplus and \otimes , a single rule performs at the same time synchronization and interleaving. The rule is defined as follows

$$\frac{\{T_i \xrightarrow{\otimes_{j_i \in \{1, 2\}} \Theta_{i, j_i}} \otimes_{j_i \in \{1, n_i\}} \Omega_{i, j_i}\}_{i \in \{1, 2\}}}{T_1 \times T_2 \xrightarrow{\otimes_{i \in \{1, 2\}} \bar{\Theta}_{i, j_i}} \otimes_{i \in \{1, 2\}, j_i \in \{1, n_i\}} \bar{\Omega}_{i, j_i}} \times$$

where $\bar{\Theta}_{j_1, j_2}^\times$ is constrained by $c_{1, j_1} \wedge c_{2, j_2}$ and its actions are all possible combinations of actions θ_{1, j_1, h_1} and θ_{2, j_2, h_2} for $\Theta_{i, j_i} = (c_{i, j_i}, \oplus_{h_i \in \{1, k_{j_i}\}} \theta_{i, j_i, h_i})$ for $i \in \{1, 2\}$. The combination of processes corresponding to the combination of actions is realized by Ω_{j_1, j_2}^\times .

Example 3. Consider an open process $T_1 \times T_2$ with $T_1 = x > 0 \nabla c?x.T, a?x.T$ and $T_2 = c!y + 1.T + a!y - 1.T$. We have $T_1 \xrightarrow{(x > 0, c?x \oplus *) \otimes (x \leq 0, a?x \oplus *)} T \oplus T_1 \otimes T \oplus T_1$ and $T_2 \xrightarrow{(true, c!y + 1 \oplus a!y - 1 \oplus *)} T \oplus T \oplus T_2$. The transition resulting from parallel composition is $T_1 \times T_2 \xrightarrow{\Theta_1 \otimes \Theta_2} \Omega_1 \otimes \Omega_2$, where $\Theta_1 = (x > 0, \tau \oplus c?z \oplus c!y + 1 \oplus a!y - 1 \oplus *)$, $\Theta_2 = (x \leq 0, \tau \oplus a?z \oplus c!y + 1 \oplus a!y - 1 \oplus *)$, $\Omega_1 = T[y + 1/x] \times T \oplus T[z/x] \times T_2 \oplus T_1 \times T \oplus T_1 \times T \oplus T_1 \times T_2$ and $\Omega_2 = T[y - 1/x] \times T \oplus T[z/x] \times T_2 \oplus T_1 \times T \oplus T_1 \times T \oplus T_1 \times T_2$. Two alternative choices corresponding to constraints $x > 0$ and $x \leq 0$ arise from composition of guard $x > 0$ with $true$ and of guard

$x \leq 0$ with *true*, respectively. For guard $x > 0$ the non-deterministic choices are obtained by the composition of actions $c?x$ and $*$ with $c!y+1, a!y-1$ and $*$, while for guard $x \leq 0$ they are obtained by the composition of actions $a?x$ and $*$ with $c!y+1, a!y-1$ and $*$. Therefore, τ actions are obtained from the synchronization of actions $c?x$ and $c!y+1$ and of $a?x$ with $a!y-1$, respectively. The others arise from interleaving by composition with $*$. The corresponding processes are obtained in the obvious way. For instance, the process corresponding to τ with guard $x > 0$ is $T[y+1/x] \times T$, since the value of $y-1$ is received by T_1 . Note that in the interleaving case with a receive action the variable x must be renamed to z to avoid clash of variables with the free variables of T_2 .

Recursive processes are handled by the classical rule, where formal parameters are substituted by actual parameters.

$$\frac{T[\bar{e}/\bar{x}] \stackrel{\otimes_{i \in \{1..n\}} \Theta_i}{\longrightarrow} \otimes_{i \in \{1..n\}} \Omega_i}{P(\bar{e}) \stackrel{\otimes_{i \in \{1..n\}} \Theta_i}{\longrightarrow} \otimes_{i \in \{1..n\}} \Omega_i} \text{rec } P(\bar{x}) \equiv T$$

Unfortunately, the application of rule *rec* leads to an infinite graph also for regular processes, where there is no parallel composition inside the scope of recursion. The semantics of [12] suffers of the same problem.

Example 4. Let us consider the process $P(x) \equiv c!x.(P(x+1) + P(x-1))$. Since the recursive process is unfolded infinitely times with a different argument an infinite graph arise.

$$\begin{aligned} P(x) &\stackrel{(true, c!x \oplus *)}{\longrightarrow} P(x+1) + P(x-1) \oplus P(x) \\ P(x+1) + P(x-1) &\stackrel{(true, c!x+1 \oplus *)}{\longrightarrow} P(x+1+1) + P(x+1-1) \oplus P(x+1) \\ P(x+1) + P(x-1) &\stackrel{(true, c!x-1 \oplus *)}{\longrightarrow} P(x-1+1) + P(x-1-1) \oplus P(x-1) \\ &\dots \end{aligned}$$

This problem can be solved by replacing in the graph transitions of $P(\bar{e})$ by transitions of a process $P(\bar{x})$ for fresh variables \bar{x} . The semantics of $P(\bar{e})$ can naturally be obtained by the semantics of $P(\bar{x})$ by instantiating parameters \bar{x} to the actual values corresponding to the evaluation of the expressions \bar{e} .

Let *general processes* \mathcal{GP} have the following syntax

$$GP ::= nil \mid a.T \mid P(\bar{x}) \mid GP_1 + GP_2 \mid be \nabla GP_1, GP_2 \mid GP_1 \times GP_2 \mid GP \setminus L$$

where \bar{x} is a tuple of distinct variables and $T \in Proc$ is a process.

Note that $a.T$ is a general process for any process T , since there are no current recursive calls. For general processes $GP \in \mathcal{GP}$, the *recursion variables* $rv(GP)$ are defined as $rv(a.T) = rv(nil) = \emptyset$, $rv(P(\bar{x})) = \{\bar{x}\}$, $rv(GP_1 + GP_2) = rv(GP_1 \times GP_2) = rv(be \nabla GP_1, GP_2) = rv(GP_1) \cup rv(GP_2)$ and $rv(GP \setminus L) = rv(GP)$.

Our aim is that of finding a general process GP that can be used instead of T in the graph. For this purpose we define *most general terms*.

Definition 5. Let T be a process. We define most general terms $\Pi(T)$:

- if $T = \text{nil}$ or $T = a.T_1$, $\Pi(T) = \{T\}$;
- for $GP_i \in \Pi(T_i)$ such that, $rv(GP_1) \cap fv(GP_2) = \emptyset$, $rv(GP_1) \cap bv(GP_2) = \emptyset$ and vice-versa, then $GP_1 + GP_2 \in \Pi(T_1 + T_2)$, $GP_1 \times GP_2 \in \Pi(T_1 \times T_2)$ and be $\nabla GP_1, GP_2 \in \Pi(\text{be } \nabla T_1, T_2)$;
- if $P(\bar{z}) \equiv T$ and T is free for $[\bar{x}/\bar{z}]$, then $P(\bar{x}) \in \Pi(P(\bar{e}))$.

Most general terms are general processes obtained by introducing fresh and distinct variables in current recursive calls. The following property is satisfied.

Proposition 6. Let T be a process. For each $GP \in \Pi(T)$, there exists a substitution σ with $\text{dom}(\sigma) = rv(GP)$ such that $GP\sigma = T$.

The substitution σ assigns actual parameters of T to formal parameters of GP . For $\rho \in Env$, let $\text{gen}_{T,GP} : Env \rightarrow Env$, such that $\rho(x) = \text{gen}_{T,GP}(\rho)(x)$, for $x \notin rv(GP)$, and $\rho(x) = S_v(\sigma(x)\rho)$, otherwise. In the environment $\text{gen}_{GP,T}(\rho)$ formal parameters of GP are instantiated to the values of the actual parameters provided by σ . The result is that the closed process $(GP, \text{gen}_{T,GP}(\rho))$ is equivalent (bisimilar) to (T, ρ) . For $p_1, p_2 \in Proc$ be processes, we say that $p_1 \equiv p_2$ iff for each $a \in Act$, for each $p_1 \xrightarrow{a} p'_1$ there exists $p_2 \xrightarrow{a} p'_2$ and $p'_1 \equiv p'_2$ and vice-versa.

Proposition 7. Let T be a process and $\rho \in Env$. For each $GP \in \Pi(T)$, $(GP, \text{gen}_{T,GP}(\rho)) \equiv (T, \rho)$.

By proposition 2 the symbolic graph where transitions of T are replaced by transitions of GP correctly models the behavior of processes.

Definition 8. Let T be a process. We define $\mathcal{SG}(T) = (GP^*, T^*, \bigotimes_{i \in \{1, n\}} \Theta_i)$ with $GP^* \subseteq \mathcal{GP}$, $T^* \subseteq Proc$ and transitions are $GP \xrightarrow{\bigotimes_{i \in \{1, n\}} \Theta_i} \bigotimes_{i \in \{1, n\}} \Omega_i$ for each $GP \in GP^*$, where

1. for each $T' \in T^*$ there exists $GP' \in GP^* \cap \Pi(T')$;
2. $T \in T^*$ and for each $GP \xrightarrow{\bigotimes_{i \in \{1, n\}} \Theta_i} \bigotimes_{i \in \{1, n\}} \Omega_i$, where $\Omega_i = \bigoplus_{j_i \in \{1, n_i\}} T_{i, j_i}$, $T_{i, j_i} \in T^*$, for $i \in \{1, n\}$, $j_i \in \{1, n_i\}$.

Example 9. Consider for instance the process $P(1)$, where $P(x)$ is defined in example 4. Since $P(z) + P(w) \in \Pi(P(x+1) + P(x-1))$ then $\mathcal{SG}(P(1))$ is finite:

$$\begin{aligned} P(x) &\xrightarrow{(true, clx \oplus *)} P(x+1) + P(x-1) \oplus P(x) \\ P(z) + P(w) &\xrightarrow{(true, clz \oplus clw \oplus *)} P(z+1) + P(z-1) \oplus P(w+1) + P(w-1) \oplus P(z) + P(w) \end{aligned}$$

This graph correctly describes the behavior of $P(1)$. For instance, the concrete computation $P(1) \xrightarrow{cl1} P(1+1) + P(1-1) \xrightarrow{cl2} P(1+1+1) + P(1+1-1) \dots$ is simulated by $(P(x), \rho_1) \xrightarrow{cl1} (P(z) + P(w), \rho_2) \xrightarrow{cl2} \dots$, where $\rho_1(x) = 1$ and $\rho_2(z) = 2$, $\rho_2(w) = 0$. Environment $\rho_2 = \text{gen}_{T,GP}(\rho_1)$ assigns to parameters z and w the result of the evaluation of expressions $x+1$ and $x-1$ with respect to ρ_1 . Note that in $P(z) + P(w)$ distinct fresh variables are used to model all recursive calls $P(e_1) + P(e_2)$, where e_1 may be different from e_2 .

The equivalence is formally stated by the following theorem. Let $K \in \{\tau, c!V, c?V\}$ and $\theta \in \text{SymAct}$. We define the constraint $\theta \in K \in \mathcal{C}$ as $\theta \in K \equiv \text{true}$, for $K = \theta = \tau$, $\theta \in K \equiv \text{true}$, for $\theta = c?x$ and $K = c?V$, $\theta \in K \equiv e \in V$, for $\theta = c!e$ and $K = c!V$, and $\theta \in K \equiv \text{false}$, otherwise. For $K = a$ we denote by $a = \theta$ the constraint $\theta \in a$.

Theorem 10. 1. For each symbolic transition $GP \xrightarrow{\otimes_{i \in \{1, n\}} \Theta_i} \otimes_{i \in \{1, n\}} \Omega_i$, where $\Theta_i = (c_i, \otimes_{j_i \in \{1, n_i\}} \theta_{i, j_i})$ and $\Omega_i = \otimes_{j_i \in \{1, n_i\}} T_{i, j_i}$ and for each $\rho \in \text{Env}$, there exists one and only one $i \in \{1, n\}$, such that $\rho \models c_i$ and, for each $j_i \in \{1, n_i\}$, there exists $a \in \text{Act}$ with $\rho \models a = \theta_{i, j_i}$ such that, $(GP, \rho) \xrightarrow{c?v} (T_{i, j_i}, \rho[x \rightarrow v])$ for $\theta_{i, j_i} = c?x$ and $v \in \text{Val}$, and $(T, \rho) \xrightarrow{a} (T_{i, j_i}, \rho)$, for $\theta_{i, j_i} \in \{\tau, c!e\}$;

2. for each $p \in P^*$ there exists $GP \in GP^*$ and $\rho \in \text{Env}$ such that $GP\rho \equiv p$ and, for each $p \xrightarrow{a} p'$, there exist $i \in \{1, n\}$ and $j_i \in \{1, n_i\}$, where $GP \xrightarrow{\otimes_{i \in \{1, n\}} \Theta_i} \otimes_{i \in \{1, n\}} \Omega_i$ with $\Theta_i = (c_i, \otimes_{j_i \in \{1, n_i\}} \theta_{i, j_i})$ and $\Omega_i = \otimes_{j_i \in \{1, n_i\}} T_{i, j_i}$, and $\rho \models c_i \wedge a = \theta_{i, j_i}$ and $p' \equiv (T_{i, j_i}, \rho[x \rightarrow v])$, for $a = c?v$ and $\theta_{i, j_i} = c?x$, and $p' \equiv (T_{i, j_i}, \rho)$, for $a \in \{\tau, c!v\}$.

By theorem [10](#) the concrete semantics is safely represented by the symbolic one and in addition concrete non-deterministic and alternative choices are exactly composed by \oplus and \otimes in symbolic transitions. Since transitions are restricted to general processes, infinite applications of *rec* with different arguments are avoided and the symbolic graph is finite for regular processes.

Theorem 11. Let $p \in \text{Proc}$ be a regular process. $\mathcal{SG}(p)$ is a finite graph up to renaming.

5 Symbolic Model Checking

In this section we show that model checking can be realized by interpreting the symbolic graph over concrete environments. Let $(\mathcal{P}(D), \subseteq)$, with $D = \{(T, \rho) \mid T \in GP^* \text{ and } \rho \in \text{Env}\}$ for $\mathcal{SG}(p) = (GP^*, T^*, \xrightarrow{\otimes_{i \in \{1, n\}} \Theta_i})$. The semantics of a formula A is defined as in Sect. [2](#) by replacing evaluations with symbolic evaluations $\delta : \text{VAR} \rightarrow \mathcal{P}(D)$ and by taking the following function for the next modality. For $K \in \{\tau, c!V, c?V\}$ and $S \in \mathcal{P}(D)$,

$$\begin{aligned} - \ll K \gg^S(S) = \{(T, \rho) \mid T \xrightarrow{\otimes_{i \in \{1, n\}} \Theta_i} \otimes_{i \in \{1, n\}} \Omega_i, \text{ with } \Theta_i = (c_i, \oplus_{j_i \in \{1, n_i\}} \theta_{i, j_i}) \text{ and } \Omega_i = \oplus_{j_i \in \{1, n_i\}} T_{i, j_i}, \text{ and there exist } i \in \{1, n\} \text{ and } j_i \in \{1, n_i\}, \\ \text{such that } \rho \models c_i \wedge \theta_{i, j_i} \in K \text{ and } (T_{i, j_i}, \rho) \in S, \text{ for } \theta_{i, j_i} \in \{\tau, c!e\}, \text{ and } \\ (T_{i, j_i}, \rho[x \rightarrow v]) \in S, \text{ for some } v \in V \text{ for } K = c?V \text{ and } \theta_{i, j_i} = c?x\}. \end{aligned}$$

where $(T, \rho) \in S$ if $(GP, \text{gen}_{T, GP}(\rho)) \in S$ for some $GP \in \Pi(T)$.

The definition of $\ll a \gg^S$ is based on the observation that process (T, ρ) is able to perform an action a , if the environment satisfies a guard c_i for which there exists a symbolic action θ_{i, j_i} corresponding to a . Moreover, the resulting process must be in S . However, since S is a set of general processes and T_{i, j_i} is not a general process we look for an equivalent process $(GP, \text{gen}_{T_{i, j_i}, GP}(\rho)) \in S$.

Theorem 12. *For each closed formula A , $\| A \|^S \cap P^* = \| A \|^S$.*

6 Abstract Model Checking

In this section we define the approximate semantics of the *collecting semantics* $\| A \|^S$ on the abstract domain obtained by replacing concrete environments with abstract environments on an abstract values domain.

Let (α_v, γ_v) be a Galois insertion between the complete lattices $(\mathcal{P}(Val), \subseteq)$ and $(\mathcal{P}(Val^\#), \subseteq)$ of concrete and abstract values. We consider the set of *abstract environments* $Env^\# = \{\rho^\# \mid \rho^\# : Var \rightarrow Val^\#\}$ and the set of *abstract processes* $D^\# = \{(T, \rho^\#) \mid T \in GP^* \text{ and } \rho^\# \in Env^\#\}$ for $\mathcal{SG}(p) = (GP^*, T^*, \bigotimes_{i \in \{1..n\}} \Theta_i)$. Let $\gamma_e : \mathcal{P}(Env^\#) \rightarrow \mathcal{P}(Env)$ and $\gamma : \mathcal{P}(D^\#) \rightarrow \mathcal{P}(D)$ be the obvious functions induced by γ_v .

Our purpose is that of computing a lower approximation of the semantics $\| A \|^l$, such that $\gamma(\| A \|^l) \subseteq \| A \|^S$. Given the relation between concrete and abstract domain an approximate semantics can be obtained by replacing concrete functions with corresponding safe abstract functions. However, the operator of negation is not monotonic. Kelb [13] argues that a lower approximation of the full logic with negation can be obtained by combining dual approximations for formulas without negation $\| A \|^l$ and $\| A \|^u$. In fact, $\| \neg A \|^l = D^\# \setminus \| A \|^u$ is a safe lower approximation of $\| \neg A \|^S$, since $\| A \|^S \subseteq \gamma(\| A \|^u)$. Analogously, $\| \neg A \|^u = D^\# \setminus \| A \|^l$ is a safe upper approximation of $\| \neg A \|^S$. Thus, the problem is reduced to the definition of safe dual approximations for all the logical operators except negation. The safeness of the dual approximations is formally defined with respect to the following framework.

Proposition 13. *There exist $\alpha^l, \alpha^u : \mathcal{P}(D) \rightarrow \mathcal{P}(D^\#)$ such that (α^u, γ) is a Galois insertion between $(\mathcal{P}(D), \subseteq)$ and $(\mathcal{P}(D^\#), \subseteq)$ and (α^l, γ) is a Galois insertion between $(\mathcal{P}(D), \supseteq)$ and $(\mathcal{P}(D^\#), \supseteq)$.*

We show that both the upper and the lower approximation can be computed over the symbolic graph with respect to abstract environments. We exploit the following constraints based on relations \oplus and \otimes . The symbol \vee denotes with an abuse of notation the equivalent constraint.

Definition 14. *Let $T \xrightarrow{\bigotimes_{i \in \{1..n\}} \Theta_i} \bigotimes_{i \in \{1..n\}} \Omega_i$, with $\Theta_i = (c_i, \bigotimes_{j_i \in \{1..n_i\}} \theta_{i,j_i})$ and $\Omega_i = \bigotimes_{j_i \in \{1..n_i\}} T_{i,j_i}$, be a symbolic transition. For each $K \in \{\tau, c!V, c?V\}$ and $i \in \{1, n\}$, $j_i \in \{1, n_i\}$ and $I = \bigcup_{i \in \{1..n\}} I_i$ with $I_i \subseteq \{1, n_i\}$ we define*

1. $FREE_{K,i,j_i} \equiv c_i \wedge \theta_{i,j_i} \in K$;
2. $CON_{K,I} \equiv \bigwedge_{i \in \{1..n\}} \neg c_i \vee (\bigvee_{j_i \in I_i} \theta_{i,j_i} \in K)$.

The abstract semantics is based on safe lower and upper approximations of the previous constraints and on dual approximations of the relation \equiv . Note that for each $c \in \mathcal{C}$, $\| c \|^u$ and $\| c \|^l$ are safe approximations iff $\| c \|^S \subseteq \gamma_e(\| c \|^u)$ and $\gamma_e(\| c \|^l) \subseteq \| c \|^S$, respectively. Moreover, \equiv is a safe upper approximation,

if $\rho \in \gamma_e(\rho^\#)$ such that $(T, \rho) \in \equiv \gamma(S^\#)$ implies $(T, \rho^\#) \in^u S^\#$, while \in^l is a safe lower approximation if $(T, \rho^\#) \in^l S^\#$ only if, for all $\rho \in \gamma_e(\rho^\#)$, $(T, \rho) \in \equiv \gamma(S^\#)$.

Let $\delta^\# : VAR \rightarrow \mathcal{P}(D^\#)$ be an abstract evaluation. We define the abstract upper and lower semantics of an open formula A with respect to $\delta^\#$ as follows.

Lower Approximation

$$\begin{aligned} \|X\|_{\delta^\#}^l &= \delta^\#(X) & \| \mu X. A \|_{\delta^\#}^l &= \mu V. (\|A\|_{\delta^\#}^l[V/X]) \\ \|A_0 \wedge A_1\|_{\delta^\#}^l &= \|A_0\|_{\delta^\#}^l \cap \|A_1\|_{\delta^\#}^l & \| \neg A \|_{\delta^\#}^l &= D^\# \setminus (\|A\|_{\delta^\#}^l) \\ \|<K>A\|_{\delta^\#}^l &= \|<K>\|_{\delta^\#}^l (\|A\|_{\delta^\#}^l) \end{aligned}$$

where, for $S^\# \in \mathcal{P}(D^\#)$,

$$\begin{aligned} - \|<K>\|_{\delta^\#}^l (S^\#) &= \{(T, \rho^\#) \mid T \xrightarrow{\otimes_{i \in \{1, n\}} \Theta_i} \otimes_{i \in \{1, n\}} \Omega_i, \text{ with } \Theta_i = (c_i, \otimes_{j_i \in \{1, n_i\}} \theta_{i, j_i}) \text{ and } \Omega_i = \otimes_{j_i \in \{1, n_i\}} T_{i, j_i}, \text{ and there exists } I = \cup_{i \in \{1, n\}} I_i \text{ with } I_i \subseteq \{1, n_i\}, \text{ such that } I \text{ is minimal and } \rho^\# \in \|CON_{K, I}\|_{\delta^\#}^l, \text{ and, for each } i \in \{1, n\} \text{ and } j_i \in I_i, (T_{i, j_i}, \rho^\#) \in^l S^\#, \text{ for } \theta_{i, j_i} \in \{cle, \tau\}, \text{ and } (T_{i, j_i}, \rho^\#[x \rightarrow \alpha_v(v)]) \in^l S^\#, \text{ for } v \in V, K = c?V \text{ and } \theta_{i, j_i} = c?x\} \}. \end{aligned}$$

Upper Approximation

$$\begin{aligned} \|X\|_{\delta^\#}^u &= \delta^\#(X) & \| \mu X. A \|_{\delta^\#}^u &= \mu V. (\|A\|_{\delta^\#}^u[V/X]) \\ \|A_0 \wedge A_1\|_{\delta^\#}^u &= \|A_0\|_{\delta^\#}^u \cap \|A_1\|_{\delta^\#}^u & \| \neg A \|_{\delta^\#}^u &= D^\# \setminus (\|A\|_{\delta^\#}^l) \\ \|<K>A\|_{\delta^\#}^u &= \|<K>\|_{\delta^\#}^u (\|A\|_{\delta^\#}^u) \end{aligned}$$

where, for $S^\# \in \mathcal{P}(D^\#)$,

$$\begin{aligned} - \|<K>\|_{\delta^\#}^u (S^\#) &= \{(T, \rho^\#) \mid T \xrightarrow{\otimes_{i \in \{1, n\}} \Theta_i} \otimes_{i \in \{1, n\}} \Omega_i \text{ with } \Theta_i = (c_i, \oplus_{j_i \in \{1, n_i\}} \theta_{i, j_i}) \text{ and } \Omega_i = \oplus_{j_i \in \{1, n_i\}} T_{i, j_i}, \text{ and there exist } i \in \{1, n\} \text{ and } j_i \in \{1, n_i\} \text{ such that } \rho^\# \in \|FREE_{K, i, j_i}\|_{\delta^\#}^u \text{ and } (T_{i, j_i}, \rho^\#) \in^u S^\#, \text{ for } \theta_{i, j_i} \in \{\tau, cle\}, \text{ and } (T_{i, j_i}, \rho^\#[x \rightarrow \alpha_v(v)]) \in^u S^\#, \text{ for } v \in V, K = c?V \text{ and } \theta_{i, j_i} = c?x\} \}. \end{aligned}$$

Let us explain the dual approximations of the next modality.

1. The *upper approximation* is safe, if $(T, \rho) \in \|<a>\|_{\delta^\#}^S (\gamma(S^\#))$ implies $(T, \rho^\#) \in \|<a>\|_{\delta^\#}^u (S^\#)$ for $\rho \in \gamma_e(\rho^\#)$. In other words, safeness requires to consider at least the abstract transitions corresponding to the concrete a -transitions (*free transition relation* [10,4]). If constraint $FREE_{a, i, j_i} \equiv c_i \wedge \theta_{i, j_i} = a$ is safely upper approximate the previous condition is guaranteed, since $\rho \models c_i \wedge \theta_{i, j_i} = a$ implies $\rho^\# \in \|c_i \wedge \theta_{i, j_i} = a\|_{\delta^\#}^u$.
2. The definition of the *lower approximation* is quite complex. In this case safeness is ensured, only if $(T, \rho^\#) \in \|<a>\|_{\delta^\#}^l (S^\#)$ implies $(T, \rho) \in \|<a>\|_{\delta^\#}^S (\gamma(S^\#))$, for each $\rho \in \gamma_e(\rho^\#)$. In other words, safeness requires to consider only the abstract transitions, for which all corresponding concrete a -transition exist (*constrained transition relation* [10,4]). Let us consider the

constraint $CON_{a,I} \equiv \bigwedge_{i \in \{1,n\}} (c_i \supset (\bigvee_{j_i \in I_i} (a = \theta_{i,j_i})))$ and its lower approximation. If $\rho^\# \in \ll CON_{a,I} \rr^l$, then for each $\rho \in \gamma_e(\rho^\#)$, if $\rho \models c_i$ there exists an action θ_{i,j_i} with $j_i \in I_i$ corresponding to a . By theorem [11](#) for each ρ there exists exactly one c_i such that $\rho \models c_i$ so that, for each ρ there exists indeed a transition with action a .

Note that by definition of the next modality safeness for both approximations requires in addition safeness of the dual approximations of \in_{\equiv} to check the resulting processes to be in $S^\#$.

Example 15. Consider a process $T = x > 0 \nabla c?x.T_1, a?x.T_2$, whose behavior is described by $T \xrightarrow{(x>0, c?x \oplus *) \otimes (x \leq 0, a?x \oplus *)} T_1 \oplus T \otimes T_2 \oplus T$. Let the values abstraction be $\alpha_v(n) = \bullet$ and let $\rho^\#$ be the abstract environment with $\rho^\#(x) = \bullet$. For a formula $A \equiv \ll c?n > true \rr$ we obtain an upper approximation $\{(T, \rho^\#)\} = \ll A \rr^u$. In fact, there exists $\rho \in \gamma_e(\rho^\#)$ such that $\rho \models x > 0 \wedge (c?x = c?n) \equiv x > 0 \wedge true$ so that $\rho^\# \in \ll x > 0 \wedge true \rr^u$. Analogously, $\{(T, \rho^\#)\} = \ll < a?n > true \rr^u$. The abstract operator is safe, since the existence of both concrete transitions $(T, \rho_1) \xrightarrow{c?n} (T_1[n/x], \rho_1)$ and $(T, \rho_2) \xrightarrow{a?n} (T_2[n/x], \rho_2)$ for $\rho_1, \rho_2 \in \gamma_e(\rho^\#)$ is captured. Note that in the abstract process non-determinism among the two actions $c?n$ and $a?n$ arise even if in the concrete case these are two alternative choices.

In contrast, let us consider the lower approximation for formula A . Since there exists $\rho_1, \rho_2 \in \gamma_e(\rho^\#)$ such that $(T, \rho_1) \not\xrightarrow{a?n}$ and $(T, \rho_2) \not\xrightarrow{c?n}$ the lower approximation is safe if and only if $(T, \rho^\#) \notin \ll A \rr^l$. Since $\ll x > 0 \rr^l = \emptyset$, $\ll x \leq 0 \rr^l = \emptyset$ and $c?n = a?x \equiv false$, $\rho^\# \notin \ll (x \geq 0 \vee true) \wedge (x < 0 \vee false) \rr^l$. Therefore, we have both $\ll < c?n > true \rr^l = \emptyset$ and $\ll < a?n > true \rr^l = \emptyset$. The abstract operator is able to observe that there is no real non-determinism between $a?n$ and $c?n$, while this is an alternative choice as expressed by \otimes and there are concrete processes for both alternatives.

On the other hand, consider the values abstraction with $\alpha_v(n) = Pos$, if $n > 0$, and $\alpha_v(n) = Neg$, otherwise. There are two abstract environments $\rho_1^\#$, with $\rho_1^\#(x) = Pos$, and $\rho_2^\#$, with $\rho_2^\#(x) = Neg$. Since for each $\rho \in \gamma(\rho_1^\#)$, $\rho \in \ll c \rr$ for $c = (x > 0 \supset true) \wedge (x \leq 0 \supset false)$, then $\rho_1^\# \in \ll c \rr^l$ is safe. With this evaluation of constraint we can safely obtain $\ll < c?n > true \rr^l = \{(T, \rho_1^\#)\}$. Since, for each concrete environment of $\rho_1^\#$ only the alternative $x > 0$ is possible, it is safe to conclude that all concrete processes indeed perform $c?n$.

Lemma 16. *Let $\ll c \rr^l$ and $\ll c \rr^u$ be safe lower and upper approximations for $c \in \mathcal{C}$. Moreover, let \in_{\equiv}^l and \in_{\equiv}^u be safe lower and upper approximations of \in_{\equiv} . For each $S^\# \in \mathcal{P}(D^\#)$ and $K \in \{\tau, c?V, c!V\}$, $\alpha^u(\ll < K > \rr^S(\gamma(S^\#))) \subseteq \ll < K > \rr^u(S^\#)$ and $\ll < K > \rr^l(S^\#) \subseteq \alpha^l(\ll < K > \rr^S(\gamma(S^\#)))$.*

By lemma 1 the lower approximation of the full logic is safe.

Theorem 17. *Let $\ll c \rr^l$ and $\ll c \rr^u$ be safe lower and upper approximations for $c \in \mathcal{C}$. Moreover, let \in_{\equiv}^l and \in_{\equiv}^u be safe lower and upper approximations of \in_{\equiv} . For each closed μ -calculus formula A , $\ll A \rr^S \supseteq \ll A \rr^l$.*

7 About Optimality

We have defined a method for constructing safe dual approximations of the next modality, that exploit dual approximations of constraints and of relation \in_{\equiv} . In this setting precision of abstract model checking depends on precision of these approximations. An interesting problem is that of finding conditions on the approximations of constraints and of \in_{\equiv} , that guarantee optimality of next modalities. It turns out that optimality of the upper approximation of constraint $FREE_{K,i,j_i}$ and of the lower approximation of constraint $CON_{K,I}$ is sufficient, whenever $\mathcal{SP}(p)$ contains only general processes.

Lemma 18. *Let p be a closed process such that $\mathcal{SP}(p) = (GP^*, T^*, \bigotimes_{i \in \{1..n\}} \theta_i)$ where $GP^* = T^*$. Let $\|c\|^u$ and $\|c\|^l$ be optimal upper and lower approximations for $c \in \mathcal{C}$. For each $S^\# \in \mathcal{P}(D^\#)$ and $K \in \{\tau, c?V, c!V\}$, $\alpha^u(\|<K>\|^S(\gamma(S^\#))) \supseteq \|<K>\|^u(S^\#)$ and $\alpha^l(\|<K>\|^S(\gamma(S^\#))) \subseteq \|<K>\|^l(S^\#)$.*

In the upper approximation case this result is quite obvious, since by optimality $\rho^\# \in \|FREE_{a,i,j_i}\|^u$ implies the existence of an environment such that $\rho \in \|FREE_{a,i,j_i}\|$, namely the existence of a corresponding concrete transition. The lower approximation is optimal, whenever there exists a constrained transition if and only if for each concrete process there exists a corresponding concrete transition. Suppose that for each concrete process $(T, \rho) \xrightarrow{a} (T', \rho')$. For each ρ there exists a choice such that constraint c_i is satisfied and a non-deterministic choice such that $\rho \models \theta_{i,j_i} = a$. Optimality is guaranteed, since theorem 1 ensures in addition that for each ρ all others alternatives c_j are not true. Therefore, for each environment the constraint $CON_{a,I}$ is indeed satisfied so that by optimality the corresponding abstract transition is certainly considered.

In contrast, if $\mathcal{SP}(p)$ contains non-general processes lemma 2 is no longer valid. Problems arise from the abstract evaluation of parameters included in the definition of \in_{\equiv} . It is sufficient to consider the case of $\rho^\#$, such that $\exists \rho_1, \rho_2 \in \gamma_e(\rho^\#)$ such that $\rho_1 \models FREE_{a,i,j_i}$ and $(T_{i,j_i}, \rho_2) \in_{\equiv} \gamma(S^\#)$, but $\rho_1 \neq \rho_2$.

Depending on the domain of values and expressions specific solutions must be studied for approximating constraints and \in_{\equiv} . We suggest a general strategy. Safe dual approximations of constraints can be found on the basis of dual approximations of basic constraints be , $e = e$ and $e \in V$ and by combining lower and upper approximations in the obvious way.

Definition 19. *For $c \in \mathcal{C}$, let $\| \neg c \|^u = \mathcal{P}(Env^\#) \setminus \| \neg c \|^l$, $\| \neg c \|^l = \mathcal{P}(Env^\#) \setminus \| \neg c \|^u$, $\| c_1 \wedge c_2 \|^l = \cap_{i \in \{1,2\}} \| c_i \|^l$ and $\| c_1 \wedge c_2 \|^u = \cap_{i \in \{1,2\}} \| c_i \|^u$.*

If basic constraints are safely approximate these approximations are obviously safe. Unfortunately, they are in general non-optimal, since α^u does not preserve \cap , while α^l does not preserve \cup .

Moreover, we have to compute dual approximations of \in_{\equiv} , that realizes parameters evaluation. Let $gen_{T,GP}^\# : Env^\# \rightarrow \mathcal{P}(Env^\#)$ be a safe approximation of $gen_{T,GP}$, where $gen_{T,GP}(\rho) \in \gamma_e(gen_{T,GP}^\#(\rho^\#))$ for each $\rho \in \gamma_e(\rho^\#)$. This function can be suitably used to define approximations of \in_{\equiv} .

Definition 20. Let $gen_{T,GP}^\#$ be a safe approximation. We define

1. $(T, \rho^\#) \in \equiv^u S^\#$ iff there exists $\rho_1^\# \in gen_{T,GP}^\#(\rho^\#)$ such that $(GP, \rho_1^\#) \in S^\#$;
2. $(T, \rho^\#) \in \equiv^l S^\#$ iff for each $\rho_1^\# \in gen_{T,GP}^\#(\rho^\#)$, $(GP, \rho_1^\#) \in S^\#$.

If $gen_{T,GP}^\#$ is safe, then the previous approximations of $\in \equiv$ are safe.

The difficulties for computing optimal dual approximations are obvious. However, it is important to stress the essential role of \otimes and \oplus for limiting the loss of information the lower approximation. The relations \otimes and \oplus allows to improve the precision of the lower approximation with respect to the “trivial” definition, where a lower approximation of constraint $FREE_{K,i,j_i}$ is considered.

Example 21. Consider the process $T_1 \times T_2$ with $T_1 = x > 0 \nabla c?x.T, a?x.T$ and $T_2 = c!y+1.T+a!y-1.T$ of example 3. With respect to the abstraction $\alpha_v(n) = \bullet$, for each $n \in Nat$, we trivially obtain $\rho^\# \in \parallel^l c$ for $c = (x > 0 \supset (\tau = \tau)) \wedge (x \leq 0 \supset (\tau = \tau))$ and $\rho^\#(x) = \bullet$, since $\tau = \tau \equiv true$. Therefore, $(T_1 \times T_2, \rho^\#) \in \parallel^l \tau > true$ is established. The lower approximation of constraint c captures that there are processes for both alternatives, but in both cases the action τ can be performed. In other words, a constrained transition $(T_1 \times T_2, \rho^\#) \xrightarrow{\tau} \dots$ is constructed. If we consider the trivial definition of lower approximation, we would not be able to prove it, since neither $\rho^\# \in \parallel (x > 0 \wedge (\tau = \tau))$ nor $\rho^\# \in \parallel (x \leq 0 \wedge (\tau = \tau))$, even if the evaluation of constraints is optimal. Thus, this trivial method succeeds only if the alternative choice is the same for each concrete processes. In contrast, due to \otimes and \oplus a weaker condition can be considered and a more precise result is achieved. The proposed method could not give the same results on a classical symbolic semantics, where branching represents both non-deterministic and alternative choices, since it exploits the existence of exactly one alternative choice for each process and the representation of all non-deterministic choices for each possibility.

8 Related Works

The combination of abstract interpretation and model checking has been the topic of intensive research in the last few years. Much of the work concerned the definition of safe abstract model, namely of safe abstract transition relations, that preserves universal properties [23,117] and both universal and existential properties [110,134,15]. [110,4] propose the use of constrained and free transitions to handle branching modalities and [10] tackles also the problem of effectively computing for very simple programs an abstract model. The proposed method suffers of the problems of the trivial definition shown in example 7. A slight different approach is the one of Kelb [13], that investigates conditions for the safeness of abstract μ -calculus semantics instead that for the safeness of abstract models. In order to handle non-monotonic negation the combination of dual approximations is suggested. However, the problem of computing safe dual approximations of the next modality even over a given abstract model is not

addressed. In the framework of value-passing concurrent processes [5] proposes an abstract labelled transition system semantics for abstract closed processes obtained by an abstraction on the values domain. It is not obvious which class of temporal properties is preserved by the abstract model. In [15] it has been introduced the idea of representing the relation of non-determinism and of alternative choice among actions in order to compute more precise safe and constrained transition relations in the framework of closed abstract processes rather than in a symbolic approach with environments. Schmidt [17] shows a methodology for computing a finite approximate semantics of value-passing CCS by finitely approximating the semantics over abstract environments as a regular tree. Such a semantics is adequate for the verification of universal properties only. As far as concern the symbolic semantics, it is worth mentioning that other approaches have been proposed for representing regular processes by finite graphs [16].

9 Conclusions

In this paper we have applied abstract interpretation to the verification of μ -calculus properties of value-passing concurrent processes. The main contribution is the definition of a finite symbolic semantics of processes, that differs from the classical one [12] in some aspects. First, classical branching of transitions is replaced by explicit relations of alternative and non-deterministic choices among transitions. Moreover, infinite branches are avoided by representing current recursive calls by means of general processes. The concrete semantics of μ -calculus can suitably be computed by interpreting the symbolic graph over concrete environments, but due to infinite values it is not effectively computable. We have proposed a technique to compute a lower approximate semantics on the symbolic graph by replacing concrete environments with abstract environments. Following the approach of [13] for explicitly treating negation the lower approximate semantics has been obtained by combining lower and upper approximations for each operator of the logic except negation. The relations of non-deterministic and alternative choices turn out to be very useful to approximate the next modality. The lower approximation in particular results undoubtedly more precise than previous proposals [11,10].

With respect to the classical approach to abstract model checking the proposed method does not rely on the construction of a safe abstract model, but on the computation of safe approximations of the “model checking” functions over the symbolic graph. This approach has several advantages. In order to prove a property it would be typically necessary to subsequently refine the chosen value abstraction by adding more information. This way the construction of the new abstract model is avoided. Moreover, this approach fits well in the traditional abstract interpretation framework and allows us to reason about safety and precision without introducing ad-hoc conditions as for instance the approximation ordering between abstract models of [4,10]. Recently, Schmidt [18] following the ideas of [19] has pointed out the very close connection among abstract model checking and data-flow analysis. These results suggest that the methods from

one area can be usefully used in the other. There are many directions to continue this research. For instance, it seems interesting to study whether the refinement operators [7] that have been designed to systematically construct new more precise domain can be applied to the abstract model checking framework.

Acknowledgments. I would like to thank Radhia and Patrick Cousot and Dave Schmidt for their helpful suggestions on the presentation of this work.

References

1. S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Proceedings of CAV 92*, volume 663 of *Lecture Notes in Computer Science*, pages 260–263. Springer-Verlag, Berlin, 1992.
2. E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. In *Proc. 19th Annual ACM Symp. on Principles of Programming Languages*, pages 343–354. ACM Press, 1992.
3. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 5(16):1512–1542, 1994.
4. R. Cleaveland, P. Iyer, and D. Yankelevic. Optimality in Abstractions of Model Checking. In *Proceedings of SAS 95*, volume 983 of *Lecture Notes in Computer Science*, pages 51–63. Springer-Verlag, Berlin, 1995.
5. R. Cleaveland and J. Riely. Testing based abstractions for value-based systems. In *Proceedings of CONCUR 94*, volume 836 of *Lecture Notes in Computer Science*, pages 417–432. Springer-Verlag, Berlin, 1994.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.
7. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.
8. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven university of Technology, 1996.
9. D. Dams, R. Gerth, and O. Grumberg. Generation of reduced models for checking fragments of *CTL*. In *Proceedings of CAV 93*, volume 697 of *Lecture Notes in Computer Science*, pages 479–490. Springer-Verlag, Berlin, 1993.
10. D. Dams, R. Gerth, and O. Grumberg. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.
11. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall CTL^*$, $\exists CTL^*$ and CTL^* . In *Proceedings of the Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 1994.
12. M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.
13. P. Kelb. Model Checking and Abstraction: A framework preserving both truth and failure information. Technical report, OFFIS, Oldenburg, Germany, 1994.
14. D. Kozen. Results on the Propositional μ -Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

15. F. Levi. Abstract model checking of value-passing processes. In A. Bossi, editor, *International Workshop on Verification, Model Checking and Abstract Interpretation*, 1997. <http://www.dsi.unive.it/~bossi/VMCAI.html>.
16. H. Lin. Symbolic Transition Graph with Assignment. In *Proc. of CONCUR 96*, volume 1119 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, Berlin, 1996.
17. D.A. Schmidt. Abstract Interpretation of Small-Step Semantics. In *Proc. of the LOMAPS Workshop on “Analysis and Verification of Multiple-Agent Languages”*, volume 1192 of *Lecture Notes in Computer Science*, pages 76–99, 1996.
18. D.A. Schmidt. Data Flow Analysis is Model Checking of Abstract Interpretation. In *Proc. of the Annual ACM Symp. on Principles of Programming Languages*, pages 38–48. ACM Press, 1998.
19. B. Steffen. Data Flow Analysis as Model Checking. In A. Meyer, editor, *Proceedings of Theoretical Aspects of Computer Software (TACS 91)*, volume 526 of *Lecture Notes in Computer Science*, pages 346–364. Springer-Verlag, Berlin, 1991.

A Appendix

Table 1. The concrete semantics

$$\tau.p \xrightarrow{\tau} p \quad c?x.p \xrightarrow{c?v} p[v/x] \quad v \in Val \quad c!e.p \xrightarrow{c!v} p \quad \mathcal{S}_v(e) = v$$

$$\frac{p_i \xrightarrow{a} p'_i}{p_1 + p_2 \xrightarrow{a} p'_1 + p'_2} + \frac{p_1 \xrightarrow{a} p'_1}{p_1 \times p_2 \xrightarrow{a} p'_1 \times p_2} \text{int}_1 \frac{p_2 \xrightarrow{a} p'_2}{p_1 \times p_2 \xrightarrow{a} p_1 \times p'_2} \text{int}_2$$

$$\frac{p_1 \xrightarrow{a} p'_1 \quad p_2 \xrightarrow{\bar{a}} p'_2}{p_1 \times p_2 \xrightarrow{\tau} p'_1 \times p'_2} \text{sync} \quad \frac{T[\bar{e}/\bar{x}] \xrightarrow{a} p}{P(\bar{e}) \xrightarrow{a} p} \text{rec} \quad P(\bar{x}) \equiv T \quad \frac{p_1 \xrightarrow{a} p'_1}{be \nabla p_1, p_2 \xrightarrow{a} p'_1} \nabla_1 \quad \mathcal{S}_b(be) = tt$$

$$\frac{p_2 \xrightarrow{a} p'_2}{be \nabla p_1, p_2 \xrightarrow{a} p'_2} \nabla_2 \quad \mathcal{S}_b(be) = ff \quad \frac{p \xrightarrow{a} p'}{p \setminus L \xrightarrow{a} p' \setminus L} \quad \text{chan}(a) \cap L = \emptyset$$

The semantic rules for the symbolic semantics are based on some operations over actions. Let $\Theta_{i,j_i} = (c_{i,j_i}, \theta_{i,j_i,1} \oplus \dots \oplus \theta_{i,j_i,k_{j_i}})$ and $\Omega_{i,j_i} = T_{i,j_i,1} \oplus \dots \oplus T_{i,j_i,k_{j_i}}$, for $i \in \{1, 2\}$, $j_i \in \{1, n_i\}$, we define

- $\Theta_{j_1,j_2}^+ = (c_{1,j_1} \wedge c_{2,j_2}, \oplus_{h_1 \in \{K_1\}} \theta_{1,j_1,h_1} \oplus \oplus_{h_2 \in \{1,K_2\}} \theta_{1,j_2,h_2} \oplus \{*\})$ and $\Omega_{j_1,j_2}^+ = \oplus_{h_1 \in \{K_1\}} T_{1,j_1} \oplus \oplus_{h_2 \in \{1,K_2\}} T_{1,j_2} \oplus T_1 + T_2$, for $K_i = \{h_i \in \{1, k_{j_i}\} \mid \theta_{i,j_i,h_i} \neq *\}$;
- $\bar{\Theta}_{1,j_1} = (be \wedge c_{1,j_1}, \bar{\theta}_{1,j_1,1} \oplus \dots \oplus \bar{\theta}_{1,j_1,k_{j_1}})$ and $\bar{\Theta}_{2,j_2} = (\neg be \wedge c_{2,j_2}, \bar{\theta}_{2,j_2,1} \oplus \dots \oplus \bar{\theta}_{2,j_2,k_{j_2}})$, and $\bar{\Omega}_{i,j_i} = \oplus_{h_i \in \{1,k_{j_i}\}} \bar{T}_{i,j_i,h_i}$ with $\bar{T}_{i,j_i,h_i} = be \nabla T_1, T_2$ if $\theta_{i,j_i,h_i} = *$ and $\bar{T}_{i,j_i,h_i} = T_{i,j_i,h_i}$ otherwise;
- $\Theta_{j_1,j_2}^\times = (c_{1,j_1} \wedge c_{2,j_2}, \oplus_{h_i \in \{1,k_{j_i}\}} \theta_{1,j_1,h_1} \times \theta_{2,j_2,h_2})$ and $\Omega_{j_1,j_2}^\times = \oplus_{h_i \in \{1,k_{j_i}\}} T_{1,j_1,h_1} \times T_{2,j_2,h_2}$ where

1. if $\theta_{1,j_1,h_1} \in \{\tau, c!e\}$ and $\theta_{2,j_2,h_2} = *$, then $\theta_{1,j_1,h_1} \times \theta_{2,j_2,h_2} = \theta_{1,j_1,h_1}$, $T_{1,j_1,h_1} = T'_{1,j_1,h_1}$ and $T'_{2,j_2,h_2} = T_2$ (**interleaving**);
 2. if $\theta_{1,j_1,h_1} = c?x$ and $\theta_{2,j_2,h_2} = *$ then $\theta_{1,j_1,h_1} \times \theta_{2,j_2,h_2} = c?z$, such that T_{1,j_1,h_1} is free for $[z/x]$ and T_2 is free for z . Moreover, $T'_{1,j_1,h_1} = T_{1,j_1,h_1}[z/x]$ and $T'_{2,j_2,h_2} = T_2$ (**interleaving**);
 3. if $\theta_{1,j_1,h_1} = \theta_{2,j_2,h_2} = *$, then $\theta_{1,j_1,h_1} \times \theta_{2,j_2,h_2} = *$ and $T'_{1,j_1,h_1} = T_1$ and $T'_{2,j_2,h_2} = T_2$ (**idle action**);
 4. $\theta_{1,j_1,h_1} = c!e$ and $\theta_{2,j_2,h_2} = c?x$, then $\theta_{1,j_1,h_1} \times \theta_{2,j_2,h_2} = \tau$ and $T'_{1,j_1,h_1} = T_{1,j_1,h_1}$ and $T'_{2,j_2,h_2} = T_{2,j_2,h_2}[e/x]$ (**synchronization**);
 5. symmetric rules;
- for $\Theta_i = (c_i, \theta_{i,1} \oplus \dots \oplus \theta_{i,n_i})$ and $\Omega_i = T_{i,1} \oplus \dots \oplus T_{i,n_i}$, $\Theta_i^\setminus = (c_i, \oplus_{j_i \in K_i} \theta_{i,j_i})$ and $K_i = \{j_i \in \{1, n_i\} \mid \text{chan}(\theta_{i,j_i}) \cap L = \emptyset \text{ and } \Omega_i^\setminus = \oplus_{j_i \in K_i} T_{i,j_i} \setminus L$.

Table 2. The symbolic semantics

$$\begin{array}{c}
 \text{nil} \xrightarrow{(true,*)} \text{nil} \quad a.T \xrightarrow{(true, a \oplus *)} T \oplus a.T \quad a \in \{\tau, c!e, c?x\} \\
 \\
 \frac{\{T_i \xrightarrow{\otimes_{j_i \in \{1, n_i\}} \Theta_i} \otimes_{j_i \in \{1, n_i\}} \Omega_{i,j_i}\}_{i \in \{1,2\}}}{T_1 + T_2 \xrightarrow{\otimes_{i \in \{1,2\}} \otimes_{j_i \in \{1, n_i\}} \Theta_{j_1, j_2}^+} \otimes_{i \in \{1,2\}, j \in \{1, n_i\}} \Omega_{j_1, j_2}^+} + \\
 \\
 \frac{\{T_i \xrightarrow{\otimes_{j_i \in \{1, n_i\}} \Theta_i} \otimes_{j_i \in \{1, n_i\}} \Omega_{i,j_i}\}_{i \in \{1,2\}}}{be \nabla T_1, T_2 \xrightarrow{\otimes_{j_1 \in \{1, n_1\}} \bar{\Theta}_1 \otimes_{j_2 \in \{1, n_2\}} \bar{\Theta}_2} \otimes_{j_1 \in \{1, n_1\}} \bar{\Omega}_{1,j_1} \otimes_{j_2 \in \{1, n_2\}} \bar{\Omega}_{2,j_2}} \nabla \\
 \\
 \frac{\{T_i \xrightarrow{\otimes_{j_i \in \{1, n_i\}} \Theta_i} \otimes_{j_i \in \{1, n_i\}} \Omega_{i,j_i}\}_{i \in \{1,2\}}}{T_1 \times T_2 \xrightarrow{\otimes_{i \in \{1,2\}} \otimes_{j_i \in \{1, n_i\}} \Theta_{j_1, j_2}^\times} \otimes_{i \in \{1,2\}, j_i \in \{1, n_i\}} \Omega_{j_1, j_2}^\times} \times \\
 \\
 \frac{T[\bar{e}/\bar{x}] \xrightarrow{\otimes_{i \in \{1, n\}} \Theta_i} \otimes_{i \in \{1, n\}} \Omega_i}{P(\bar{e}) \xrightarrow{\otimes_{i \in \{1, n\}} \Theta_i} \otimes_{i \in \{1, n\}} \Omega_i} \text{rec } P(\bar{x}) \equiv T \quad \frac{T \xrightarrow{\otimes_{i \in \{1, n\}} \Theta_i} \otimes_{i \in \{1, n\}} \Omega_i}{T \setminus L \xrightarrow{\otimes_{i \in \{1, n\}} \Theta_i^\setminus} \otimes_{i \in \{1, n\}} \Omega_i^\setminus} \setminus
 \end{array}$$

Automatic Determination of Communication Topologies in Mobile Systems

Arnaud Venet

LIX, École Polytechnique, 91128 Palaiseau, France.

venet@lix.polytechnique.fr

<http://lix.polytechnique.fr/~venet>

Abstract. The interconnection structure of mobile systems is very difficult to predict, since communication between component agents may carry information which dynamically changes that structure. In this paper we design an automatic analysis for statically determining all potential links between the agents of a mobile system specified in the π -calculus. For this purpose, we use a nonstandard semantics of the π -calculus which allows us to describe precisely the linkage of agents. The analysis algorithm is then derived by abstract interpretation of this semantics.

Key words: π -calculus, nonstandard semantics, abstract interpretation.

1 Introduction

We are interested in analyzing the evolution of the interconnection structure, or *communication topology*, in a mobile system of processes, abstracting away all computational aspects but communication. Therefore, we can restrict our study to the π -calculus [18,19], which is a widely accepted formalism for describing communication in mobile systems. Whereas the communication topology of systems written in CSP [13] or CCS [17] can be directly extracted from the text of the specification, a semantic analysis is required in the π -calculus, because communication links may be dynamically created between agents. In the absence of automatic analysis tools this makes the design and debugging of mobile systems very difficult tasks (see [11] for a detailed case study). In this paper we propose a semantic analysis of the π -calculus based on Abstract Interpretation [35] for automatically inferring approximate but sound descriptions of communication topologies in mobile systems.

In a previous work [23] we have presented an analysis of the π -calculus which relies on a nonstandard concrete semantics. In that model recursively defined agents are identified by the sequence of replication unfoldings from which they stem, whereas the interconnection structure is given by an equivalence relation on the agent communication ports. That semantics is inspired of a representation of sharing in recursive data structures [14] which has been applied to alias analysis [10]. However, the equivalence relation does not capture an important piece of information for debugging and verification purposes: the instance of the

channel that establishes a link between two agents. In this paper we redesign our previous analysis in order to take this information into account, while still preserving a comparable level of accuracy. Surprisingly enough, whereas our original analysis was rather complicated, involving heavy operations like transitive closure of binary relations, the refined one is tremendously simpler and only requires very basic primitives.

The paper is organized as follows. In Sect. 2 we introduce our representation of mobile systems in the π -calculus. Section 3 describes the nonstandard semantics of mobile systems, which makes instances of recursively defined agents and channels explicit. The abstract interpretation gathering information on communication topologies is constructed in Sect. 4. In Sect. 5 we design a computable analysis which is able to infer accurate descriptions of unbounded and nonuniform communication topologies. Related work is discussed in Sect. 6.

2 Mobile Systems in the π -Calculus

We consider the asynchronous version of the polyadic π -calculus which was introduced by Turner [21] as a semantic basis of the PICT programming language. This restricted version has simpler communication primitives and a more operational flavour than the full π -calculus, while still ensuring a high expressive power¹. Let \mathcal{N} be a countable set of channel names. The syntax of processes is given by the following grammar:

$P ::= c![x_1, \dots, x_n]$	Message
$\quad \quad c?[x_1, \dots, x_n].P$	Input guard
$\quad \quad *c?[x_1, \dots, x_n].P$	Guarded replication
$\quad \quad (P \mid P)$	Parallel composition
$\quad \quad (\nu x)P$	Channel creation

where c, x, x_1, \dots, x_n are channel names. Input guard and channel creation act as *name binders*, i.e. in the process $c?[x_1, \dots, x_n].P$ (resp. $(\nu x)P$) the occurrences of x_1, \dots, x_n (resp. x) in P are considered bound. Usual rules about scoping, α -conversion and substitution apply. We denote by $\text{fn}(P)$ the set of *free names* of P , i.e. those names which are not in the scope of a binder.

Following the CHAM style [1], the standard semantics of the π -calculus is given by a *structural congruence* and a *reduction relation* on processes. The congruence relation “ \equiv ” satisfies the following rules:

- (i) $P \equiv Q$ whenever P and Q are α -equivalent.
- (ii) $P \mid Q \equiv Q \mid P$.
- (iii) $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$.
- (iv) $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$.
- (v) $(\nu x)P \mid Q \equiv (\nu x)(P \mid Q)$ if $x \notin \text{fn}(Q)$.

The reduction relation is defined in Fig. 1, where $P\{x_1/y_1, \dots, x_n/y_n\}$ denotes the result of substituting every name x_i for the name y_i in P . This may involve α -conversion to avoid capturing one of the x_i ’s.

¹ We can encode the lazy λ -calculus for example [2].

$$\begin{array}{c}
c![x_1, \dots, x_n] \mid c?[y_1, \dots, y_n].P \rightarrow P\{x_1/y_1, \dots, x_n/y_n\} \\
c![x_1, \dots, x_n] \mid *c?[y_1, \dots, y_n].P \rightarrow P\{x_1/y_1, \dots, x_n/y_n\} \mid *c?[y_1, \dots, y_n].P \\
\\
\frac{P \rightarrow P'}{(\nu x)P \rightarrow P'} \qquad \frac{P' \equiv P \quad P \rightarrow Q \quad Q \equiv Q'}{P' \rightarrow Q'} \qquad \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}
\end{array}$$

Fig. 1. Reduction relation in the standard semantics.

We now have to define what we mean by a “mobile system in the π -calculus”. We cannot simply allow a mobile system to be described by any process S . In fact, we are unable to design the nonstandard semantics, and hence the analysis, if we do not require S to be *closed*, i.e. $\text{fn}(S) = \emptyset$. In other words, we must consider the system in a whole. In order to make the semantic constructions of the following sections simpler, we add further constraints to the structure of mobile systems, which are inspired of Milner’s definition of *friendly systems* [18]. We denote by \mathbf{x} a tuple (x_1, \dots, x_n) of channel names. We say that a process is a *thread* if it is made of a message possibly preceded by some input guards: $c_1?[x_1] \dots c_n?[x_n].c![x]$. We call *resource* a replicated process of the following form:

$$*c?[\mathbf{x}].(\nu \mathbf{y})(T_1 \mid \dots \mid T_n)$$

where all the T_i ’s are threads. A mobile system S is then defined as:

$$S \equiv (\nu \mathbf{c})(R_1 \mid \dots \mid R_n \mid T_0)$$

where the R_i ’s are resources and T_0 is a message, the *initial thread*, which originates all communications in the system. The names in \mathbf{c} are called the *initial channels*. Therefore, all threads and channels created by a mobile system are fetched from its resources.

Example 1. We model a system S which sets up a ring of communicating processes, where each component agent may only communicate with its left and right neighbours. The system generating a ring of arbitrary size is defined as follows:

$$S \equiv (\nu \text{make})(\nu \text{mon})(\nu \text{left}_0)(R_1 \mid R_2 \mid \text{make}![\text{left}_0])$$

where

$$R_1 \equiv * \text{make}?[\text{left}].(\nu \text{right})(\text{mon}![\text{left}, \text{right}] \mid \text{make}![\text{right}])$$

is the resource that adds a new component to the chain of processes and

$$R_2 \equiv * \text{make}?[\text{left}].\text{mon}![\text{left}, \text{left}_0]$$

is the resource that closes the ring. The name “mon” should be seen as a reference to a hidden resource (for example a C program) which monitors the behaviour

of a ring component². For the sake of clarity, we denote by \mathbf{c} the free names of all agents in the system at every stage of its evolution. Then, a ring with four components can be generated by S in four steps as follows:

$$\begin{aligned}
S &\rightarrow (\nu \mathbf{c})(R_1 \mid R_2 \mid \text{mon}![\text{left}_0, \text{right}_1] \mid \text{make}![\text{right}_1]) \\
&\rightarrow (\nu \mathbf{c})(R_1 \mid R_2 \mid \text{mon}![\text{left}_0, \text{right}_1] \mid \text{mon}![\text{right}_1, \text{right}_2] \\
&\quad \mid \text{make}![\text{right}_2]) \\
&\rightarrow (\nu \mathbf{c})(R_1 \mid R_2 \mid \text{mon}![\text{left}_0, \text{right}_1] \mid \text{mon}![\text{right}_1, \text{right}_2] \\
&\quad \mid \text{mon}![\text{right}_2, \text{right}_3] \mid \text{make}![\text{right}_3]) \\
&\rightarrow (\nu \mathbf{c})(R_1 \mid R_2 \mid \text{mon}![\text{left}_0, \text{right}_1] \mid \text{mon}![\text{right}_1, \text{right}_2] \\
&\quad \mid \text{mon}![\text{right}_2, \text{right}_3] \mid \text{mon}![\text{right}_3, \text{left}_0])
\end{aligned}$$

The right_i 's represent the successive instances of the channel right created at each request to R_1 . \square

As illustrated by the above example, the configuration of a mobile system at any stage of its evolution has the following particular form:

$$(\nu \mathbf{c})(\underbrace{R_1 \mid \dots \mid R_m}_{\text{Resources}} \mid \underbrace{T_1 \mid \dots \mid T_n}_{\text{Threads}})$$

where R_1, \dots, R_m are the resources originally present in the system. Intuitively, every thread T_i or channel c_i present in the configuration could be unambiguously identified with the instant of its creation in the history of computations. Unfortunately, this precious information is not captured by the standard semantics. The process of α -conversion in particular, which is inherent to the definition of the semantics, destroys the identity of channels. The purpose of the next section is to introduce a refined semantics of the π -calculus which restores this information.

3 Nonstandard Semantics of Mobile Systems

Let S be a mobile system described in the π -calculus. In order to identify the threads and channels created by the system, we must be able to locate the syntactic components of S from which they stem. This is the role of the following notations. We denote by $R(S)$ the number of resources in S . We assume that every resource is assigned a unique number r in $\{1, \dots, R(S)\}$. For any such r , we denote by S_r the corresponding resource in S and by $T(r)$ the number of threads spawned by the resource. We similarly assign a unique number t in $\{1, \dots, T(r)\}$ to every thread in S_r and we denote by $S_{r,t}$ this thread, which has the following form:

$$S_{r,t} = c_1?[x_1] \dots c_{A(r,t)-1}?[x_{A(r,t)-1}] \cdot c_{A(r,t)}![x_{A(r,t)}]$$

² Recall that we only take communications into account, abstracting away all other computational aspects like the details of the monitoring procedure here.

where $A(r, t)$ is the number of input/output actions in $S_{r,t}$. Note that $A(r, t)$ is always nonzero because a thread contains at least one message. For $1 \leq n \leq A(r, t)$, we denote by $\text{act}(r, t, n)$ the n -th input/output action of $S_{r,t}$, and by $S_{r,t} @ n$ the subterm $c_n?[x_n] \dots c_{A(r,t)}![x_{A(r,t)}]$ of $S_{r,t}$ starting at the n -th input/output action $\text{act}(r, t, n)$. By convention, the initial thread is assigned the resource number 0. Finally, given a resource number r such that

$$S_r \equiv *c?[x].(\nu y_1) \dots (\nu y_n)(T_1 \mid \dots \mid T_{T(r)})$$

we put $\text{guard}(r) = c?[x]$ and $C(r) = \{y_1, \dots, y_n\}$.

Example 2. We consider the system of Example 1 which contains two resources R_1 and R_2 . We put $S_1 = R_1$ and $S_2 = R_2$. Thus we have:

- $\text{guard}(1) = \text{make}[left]$, $C(1) = \{right\}$, $T(1) = 2$, $S_{1,1} = \text{mon}[left, right]$, $S_{1,2} = \text{make}[right]$, $A(1, 1) = A(1, 2) = 1$.
- $\text{guard}(2) = \text{make}[left]$, $C(2) = \emptyset$, $T(2) = 1$, $S_{2,1} = \text{mon}[left, left_0]$, $A(2, 1) = 1$.

The initial thread is $S_{0,1} = \text{make}[left_0]$. □

A configuration of S in the nonstandard semantics is a finite set of *thread instances*. We do not need to represent resources since they are statically given and accessible by all the threads in any state of the system. A thread instance is a tuple $\langle r, t, n, id, E \rangle$ where $1 \leq r \leq R(S)$, $1 \leq t \leq T(r)$, $1 \leq n \leq A(r, t)$, id is a *thread identifier* and E is an *environment*. The thread identifier is the history of resource requests which led to the creation of the thread, starting from the initial one. A resource request being nothing more than a message to a replicated process, it can be identified with the thread that released this message. If we denote by

$$\text{Thr}(S) = \{(r, t) \mid 1 \leq r \leq R(S), 1 \leq t \leq T(r)\}$$

the set of all threads originally present in the system, then $id \in \text{Thr}(S)^*$. The empty sequence ε is the identifier of the initial thread. The environment E maps every free name $x \in \text{fn}(S_{r,t} @ n)$ of the thread instance to a *channel instance*. A channel instance is a tuple (r', y, id') where $1 \leq r' \leq R(S)$, $y \in C(r')$ and id' is a *channel identifier*. Instances of initial channels are represented similarly except that they are assigned the resource number 0. The channel identifier is the history of resource requests that led to the creation of the instance of channel y by resource r' . Therefore, channel identifiers and thread identifiers are represented identically, i.e. $id' \in \text{Thr}(S)^*$. Similarly, the identifier of an initial channel is the empty sequence ε .

We assume that there is no overlapping of scopes in the mobile system, i.e. we forbid terms like $x?[y].y?[y].y![z]$ or $*c?[y, z].(\nu y)z![y]$. This assumption can always be satisfied by appropriate α -conversion. The transition relation “ \triangleright ” of the nonstandard operational semantics is defined in Fig. 2 and Fig. 3. It should be clear that without the hypothesis on name scoping, the definition of the resulting environments is ambiguous. The two transition rules correspond to the

two kinds of operations that may arise in a mobile system: resource fetching and communication between threads. The initial configuration C_0 of the nonstandard semantics is given by $C_0 = \{\langle 0, 1, 1, \varepsilon, E_0 \rangle\}$ where the environment E_0 maps any $z \in \text{fn}(S_{0,1} @ 1)$ to the instance $(0, z, \varepsilon)$ of the corresponding initial channel.

Example 3. Using the notations of Example 2, a ring with four components is described in the nonstandard semantics as follows:

$$\left[\begin{array}{l} \left\langle 1, 1, 1, (0, 1), \left\{ \begin{array}{l} \text{mon} \mapsto (0, \text{mon}, \varepsilon), \\ \text{left} \mapsto (0, \text{left}_0, \varepsilon), \\ \text{right} \mapsto (1, \text{right}, (0, 1)) \end{array} \right\} \right\rangle, \\ \left\langle 1, 1, 1, (0, 1).(1, 2), \left\{ \begin{array}{l} \text{mon} \mapsto (0, \text{mon}, \varepsilon), \\ \text{left} \mapsto (1, \text{right}, (0, 1)) \\ \text{right} \mapsto (1, \text{right}, (0, 1).(1, 2)) \end{array} \right\} \right\rangle, \\ \left\langle 1, 1, 1, (0, 1).(1, 2)^2, \left\{ \begin{array}{l} \text{mon} \mapsto (0, \text{mon}, \varepsilon), \\ \text{left} \mapsto (1, \text{right}, (0, 1).(1, 2)) \\ \text{right} \mapsto (1, \text{right}, (0, 1).(1, 2)^2) \end{array} \right\} \right\rangle, \\ \left\langle 2, 1, 1, (0, 1).(1, 2)^3, \left\{ \begin{array}{l} \text{mon} \mapsto (0, \text{mon}, \varepsilon), \\ \text{left} \mapsto (1, \text{right}, (0, 1).(1, 2)^2) \\ \text{left}_0 \mapsto (0, \text{left}_0, \varepsilon) \end{array} \right\} \right\rangle \end{array} \right]$$

The generative process of the ring is now made entirely explicit thanks to the information carried by thread and channel identifiers (compare with the corresponding computation in the standard semantics given in example 1). \square

Both semantics can be shown to be equivalent by defining the translation of a nonstandard configuration C of S to a term $\pi(C)$ of the π -calculus and by applying a *bisimulation* argument. We may assume without loss of generality that the π -terms we will generate are built upon the set of names \mathcal{N}' defined as the disjoint union of \mathcal{N} with the set of all channel instances (r, y, id) that could be created by S . We identify an environment with a substitution over \mathcal{N}' . We denote by $P\sigma$ the result of applying a substitution σ to a π -term P . Then, for any nonstandard configuration

$$C = \{\langle r_1, t_1, n_1, id_1, E_1 \rangle, \dots, \langle r_k, t_k, n_k, id_k, E_k \rangle\}$$

we define the translation $\pi(C)$ as follows:

$$\pi(C) = (\nu \mathbf{c})(S_1 E'_1 \mid \dots \mid S_{R(S)} E'_{R(S)} \mid (S_{r_1, t_1} @ n_1) E_1 \mid \dots \mid (S_{r_k, t_k} @ n_k) E_k)$$

where, for $1 \leq r \leq R(S)$, E'_r is the environment which maps any $z \in \text{fn}(S_r)$ to $(0, z, \varepsilon)$. The channels in \mathbf{c} are all those which have a free occurrence in some agent of the top-level parallel composition of $\pi(C)$.

Theorem 4. If $C_0 \triangleright^* C$ and $C \triangleright C'$, then $\pi(C) \rightarrow \pi(C')$. If $C_0 \triangleright^* C$ and $\pi(C) \rightarrow P$, then there exists C' such that $C \triangleright C'$ and $P \equiv \pi(C')$.

If there are $\mu \in C$ and $1 \leq r' \leq R(S)$ such that:

- $\mu = \langle r, t, n, id, E \rangle$
- $\text{act}(r, t, n) = x![x_1, \dots, x_k]$
- $\text{guard}(r') = c?[y_1, \dots, y_k]$
- $E(x) = (0, c, \varepsilon)$

then

$$C \triangleright (C - \{\mu\}) \cup \{\langle r', t', 1, id.(r, t), E_{t'} \rangle \mid 1 \leq t' \leq T(r')\}$$

where, for all $1 \leq t' \leq T(r')$ and $z \in \text{fn}(S_{r', t'} @ 1)$

$$E_{t'}(z) = \begin{cases} E(x_i) & \text{if } z = y_i, 1 \leq i \leq k \\ (r', z, id.(r, t)) & \text{if } z \in C(r') \\ (0, z, \varepsilon) & \text{otherwise, i.e. if } z \text{ is an initial channel} \end{cases}$$

Fig. 2. Resource fetching.

A nonstandard configuration describes the communication topology of a mobile system at a particular moment of its evolution in terms of the resources initially present in the system. Therefore, the nonstandard semantics is a good basis for deriving an analysis, since the resulting information can be used to determine the role of each *syntactic* component of the system in the evolution of its interconnection structure. Constructing a computable abstraction of this semantics is the purpose of the next section.

4 Abstract Interpretation of Mobile Systems

We denote by \mathcal{C} the set of all possible nonstandard configurations for a system S . We are actually interested in the set $\mathcal{S} = \{C \in \mathcal{C} \mid C_0 \stackrel{*}{\triangleright} C\}$ of configurations of the system which are accessible from the initial one by a finite sequence of computations. This is the *collecting semantics* of S [9], which can be expressed as the least fixpoint of the \cup -complete endomorphism \mathbb{F} on the complete lattice $(\wp(\mathcal{C}), \subseteq, \cup, \emptyset, \cap, \mathcal{C})$ defined as follows:

$$\mathbb{F}(X) = \{C_0\} \cup \{C \mid \exists C' \in X : C' \triangleright C\}$$

Following the methodology of Abstract Interpretation [3, 5], we construct a lattice $(\mathcal{C}^\sharp, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, the *abstract domain*, allowing us to give finite descriptions of infinite sets of configurations. This domain is related to $\wp(\mathcal{C})$ via a monotone map $\gamma : (\mathcal{C}^\sharp, \sqsubseteq) \rightarrow (\wp(\mathcal{C}), \subseteq)$, the *concretization function*. Then we derive an abstract counterpart $\mathbb{F}^\sharp : \mathcal{C}^\sharp \rightarrow \mathcal{C}^\sharp$ of \mathbb{F} which must be *sound* with respect to γ , that is: $\mathbb{F} \circ \gamma \subseteq \gamma \circ \mathbb{F}^\sharp$.

The abstract domain \mathcal{C}^\sharp is based upon a global abstraction of all environments in a nonstandard configuration. We assume that we are provided with a lattice $(\text{Id}_2^\sharp, \sqsubseteq_2, \sqcup_2, \sqcap_2, \top_2, \bot_2)$ and a monotone map $\gamma_2 : (\text{Id}_2^\sharp, \sqsubseteq_2) \rightarrow$

If there are $\mu, \rho \in C$ such that:

- $\mu = \langle r, t, n, id, E \rangle$
- $\rho = \langle r', t', n', id', E' \rangle$
- $\text{act}(r, t, n) = x![x_1, \dots, x_k]$
- $\text{act}(r', t', n') = y?[y_1, \dots, y_k]$
- $E(x) = E'(y)$

then

$$C \triangleright (C - \{\mu, \rho\}) \cup \{\langle r', t', n' + 1, id', E'' \rangle\}$$

where, for all $z \in \text{fn}(S_{r', t'} @ n' + 1)$

$$E''(z) = \begin{cases} E(x_i) & \text{if } z = y_i, 1 \leq i \leq k \\ E'(z) & \text{otherwise} \end{cases}$$

Fig. 3. Communication between threads.

$(\wp(\text{Thr}(S)^* \times \text{Thr}(S)^*), \subseteq)$. This lattice is left as a parameter of our abstraction. It will be instantiated in the next section when we will set up an effective analysis. Let $\text{Chan}(S)$ be the set $\{(r, t, n, x, r', y) \mid (r, t) \in \text{Thr}(S), n \in A(r, t), x \in \text{fn}(S_{r, t} @ n), 1 \leq r' \leq R(S), y \in C(r')\}$ of all possible syntactic relations between a free name in a thread and a channel created by a resource of the system. The abstract domain C^\sharp is then defined as $C^\sharp = \text{Chan}(S) \rightarrow \text{Id}_2^\sharp$, the lattice operations being the pointwise extensions of those in Id_2^\sharp . Given an abstract configuration C^\sharp , $\gamma(C^\sharp)$ is the set of nonstandard configurations C such that, for any $\langle r, t, n, id, E \rangle \in C$ and any $x \in \text{fn}(S_{r, t} @ n)$, the following condition is satisfied:

$$E(x) = (r', y, id') \Rightarrow (id, id') \in \gamma_2(C^\sharp(r, t, n, x, r', y))$$

Monotonicity of γ is readily checked. For the sake of readability, we will denote an abstract configuration C^\sharp by its graph $\{\langle \varkappa_1 \mapsto id_1^\sharp \rangle, \dots, \langle \varkappa_n \mapsto id_n^\sharp \rangle\}$, where the \varkappa_i 's are in $\text{Chan}(S)$ and each id_j^\sharp is in Id_2^\sharp . We may safely omit to write pairs of the form $\langle \varkappa \mapsto \perp_2 \rangle$.

The abstract semantics is given by a transition relation \rightsquigarrow on abstract configurations. In the relation $C_1^\sharp \rightsquigarrow C_2^\sharp$, the configuration C_2^\sharp represents the *modification* to the communication topology of C_1^\sharp induced by an abstract computation. Therefore, the function \mathbb{F}^\sharp is given by:

$$\mathbb{F}^\sharp(C^\sharp) = C_0^\sharp \sqcup C^\sharp \sqcup \bigsqcup \{\bar{C}^\sharp \mid C^\sharp \rightsquigarrow \bar{C}^\sharp\}$$

where C_0^\sharp is the initial abstract configuration defined as $\{\langle 0, 1, 1, x, 0, x \rangle \mapsto \varepsilon_2 \mid x \in \text{fn}(S_{0, 1} @ 1)\}$, ε_2 being a distinguished element of Id_2^\sharp such that $(\varepsilon, \varepsilon) \in \gamma_2(\varepsilon_2)$. The transition relation \rightsquigarrow is defined in Fig. 4 and Fig. 5 by using the

If there are $\langle (r, t, n, x, 0, c) \mapsto id^\sharp \rangle \in C^\sharp$ and $1 \leq r' \leq R(S)$ such that:

- $\text{act}(r, t, n) = x![x_1, \dots, x_k]$
- $\text{guard}(r') = c?[y_1, \dots, y_k]$
- $id^\sharp \neq \perp_2$

then

$$C^\sharp \rightsquigarrow \{ \langle \varkappa \mapsto id_{t', z, r'', w}^\sharp \rangle \mid \varkappa = (r', t', 1, z, r'', w), 1 \leq t' \leq T(r') \}$$

where, for all $1 \leq t' \leq T(r')$

$$id_{t', z, r'', w}^\sharp = \begin{cases} \text{push}_{(r, t)}(C^\sharp(r, t, n, x_i, r'', w)) & \text{if } z = y_i, 1 \leq i \leq k \\ \perp_2 & \text{if } z \in C(r') \text{ and } (r'', w) \neq (r', z) \\ \text{dpush}_{(r, t)}(C^\sharp(r, t, n, x, 0, c)) & \text{if } z \in C(r') \text{ and } (r'', w) = (r', z) \\ \perp_2 & \text{if } z \text{ is initial and } (r'', w) \neq (0, z) \\ \text{spush}(C^\sharp(r, t, n, x, 0, c)) & \text{if } z \text{ is initial and } (r'', w) = (0, z) \end{cases}$$

Fig. 4. Abstract resource fetching.

following abstract primitives: a “push” operation $\text{push}_\tau : \text{Id}_2^\sharp \rightarrow \text{Id}_2^\sharp$, a “double push” $\text{dpush}_\tau : \text{Id}_2^\sharp \rightarrow \text{Id}_2^\sharp$ and a “single push” $\text{spush}_\tau : \text{Id}_2^\sharp \rightarrow \text{Id}_2^\sharp$ defined for any $\tau \in \text{Thr}(S)$, a “synchronization” operator $\text{sync} : \text{Id}_2^\sharp \times \text{Id}_2^\sharp \rightarrow \text{Id}_2^\sharp$ and a “swapping” operation $\text{swap} : \text{Id}_2^\sharp \rightarrow \text{Id}_2^\sharp$. These operations depend on the choice of Id_2^\sharp , however they must satisfy some soundness conditions:

- For any $\tau \in \text{Thr}(S)$ and $id^\sharp \in \text{Id}_2^\sharp$, $\{(id.\tau, id') \mid (id, id') \in \gamma_2(id^\sharp)\} \subseteq \gamma_2(\text{push}_\tau(id^\sharp))$.
- For any $\tau \in \text{Thr}(S)$ and $id^\sharp \in \text{Id}_2^\sharp$, $\{(id.\tau, id.\tau) \mid (id, id') \in \gamma_2(id^\sharp)\} \subseteq \gamma_2(\text{dpush}_\tau(id^\sharp))$.
- For any $\tau \in \text{Thr}(S)$ and $id^\sharp \in \text{Id}_2^\sharp$, $\{(id.\tau, \varepsilon) \mid (id, id') \in \gamma_2(id^\sharp)\} \subseteq \gamma_2(\text{spush}_\tau(id^\sharp))$.
- For any $id^\sharp \in \text{Id}_2^\sharp$, $\{(id', id) \mid (id, id') \in \gamma_2(id^\sharp)\} \subseteq \gamma_2(\text{swap}(id^\sharp))$.
- For any $id_1^\sharp, id_2^\sharp \in \text{Id}_2^\sharp$, $\{(id_1, id_2) \mid \exists id' \in \text{Thr}(S)^* : (id_1, id') \in \gamma_2(id_1^\sharp) \wedge (id_2, id') \in \gamma_2(id_2^\sharp)\} \subseteq \gamma_2(\text{sync}(id_1^\sharp, id_2^\sharp))$.

Intuitively, the push_τ operation concatenates τ to the first component of every pair of identifiers. The dpush_τ and spush_τ operations act similarly, except that dpush_τ duplicates the first component into the second one and spush_τ sets the second component to ε . The swap operation permutes the components of every pair of identifiers. The sync operation extracts pairs of identifiers corresponding to redexes, i.e. pairs of agents linked to the same instance of a channel.

Proposition 5. If $C \in \gamma(C^\sharp)$ and $C \triangleright C'$, then there exists \overline{C}^\sharp in \mathcal{C}^\sharp , such that $C^\sharp \rightsquigarrow \overline{C}^\sharp$ and $C' \in \gamma(C^\sharp \sqcup \overline{C}^\sharp)$.

The soundness of \mathbb{F}^\sharp is then a simple consequence of the previous result. It is remarkable that the soundness of the whole semantics depends only on very

If there are $\langle \varkappa_1 \mapsto id_1^\sharp \rangle, \langle \varkappa_2 \mapsto id_2^\sharp \rangle \in C^\sharp$ such that:

- $\varkappa_1 = (r, t, n, x, r'', u)$
- $\varkappa_2 = (r', t', n', y, r'', u)$
- $\text{act}(r, t, n) = x![x_1, \dots, x_k]$
- $\text{act}(r', t', n') = y?[y_1, \dots, y_k]$
- $id_s^\sharp = \text{sync}(id_1^\sharp, id_2^\sharp) \neq \perp_2$

then

$$C^\sharp \rightsquigarrow \{ \langle \varkappa \mapsto id_{z, r''', w}^\sharp \rangle \mid \varkappa = (r', t', n' + 1, z, r''', w) \}$$

where, if we put $\overline{id}_{i, r''', w}^\sharp = C^\sharp(r, n, t, x_i, r''', w)$ for $1 \leq i \leq k$

$$id_{z, r''', w}^\sharp = \begin{cases} \text{swap}(\text{sync}(\text{swap}(\overline{id}_{i, r''', w}^\sharp, \text{swap}(id_s^\sharp))) & \text{if } z = y_i, 1 \leq i \leq k \\ C^\sharp(r', t', n', z, r''', w) & \text{otherwise} \end{cases}$$

Fig. 5. Abstract communication between threads.

simple conditions over some primitive operations. This means that we only need to construct the domain Id_2^\sharp and instantiate those operations to obtain a computable and sound abstract semantics. This is the goal that we will achieve in the next section.

5 Design of a Computable Analysis

The collecting semantics \mathcal{S} is the least fixpoint of \mathbb{IF} . Therefore, by Kleene's theorem, it is the limit of the following increasing iteration sequence:

$$\begin{cases} \mathcal{S}_0 &= \emptyset \\ \mathcal{S}_{n+1} &= \mathbb{IF}(\mathcal{S}_n) \end{cases}$$

Following [3,9] we will compute a sound approximation \mathcal{S}^\sharp of \mathcal{S} by mimicking this iteration, using \mathbb{IF}^\sharp instead of \mathbb{IF} . Since the resulting computation may not terminate, we use a *widening* operator to enforce convergence in finitely many steps. A widening operator $\nabla : \mathcal{C}^\sharp \times \mathcal{C}^\sharp \rightarrow \mathcal{C}^\sharp$ must satisfy the following conditions:

- For any $C_1^\sharp, C_2^\sharp \in \mathcal{C}^\sharp$, $C_1^\sharp \sqcup C_2^\sharp \sqsubseteq C_1^\sharp \nabla C_2^\sharp$.
- For any sequence $(C_n^\sharp)_{n \geq 0}$, the sequence $(\overline{C}_n^\sharp)_{n \geq 0}$ defined as:

$$\begin{cases} \overline{C}_0^\sharp &= C_0^\sharp \\ \overline{C}_{n+1}^\sharp &= \overline{C}_n^\sharp \nabla C_{n+1}^\sharp \end{cases}$$

is ultimately stationary.

Note that we can construct a widening on \mathcal{C}^\sharp from an existing widening ∇_2 on Id_2^\sharp by pointwise application of ∇_2 . We define the approximate iteration sequence $(\mathcal{S}^\sharp_n)_{n \geq 0}$ as follows:

$$\begin{cases} \mathcal{S}^\sharp_0 &= \perp \\ \mathcal{S}^\sharp_{n+1} &= \mathcal{S}^\sharp_n \nabla \mathbb{F}^\sharp(\mathcal{S}^\sharp_n) & \text{if } \neg(\mathbb{F}^\sharp(\mathcal{S}^\sharp_n) \sqsubseteq \mathcal{S}^\sharp_n) \\ \mathcal{S}^\sharp_{n+1} &= \mathcal{S}^\sharp_n & \text{if } \mathbb{F}^\sharp(\mathcal{S}^\sharp_n) \sqsubseteq \mathcal{S}^\sharp_n \end{cases}$$

Convergence is ensured by the following result:

Theorem 6 [9]. The sequence $(\mathcal{S}^\sharp_n)_{n \geq 0}$ is ultimately stationary and its limit \mathcal{S}^\sharp satisfies $\mathcal{S} \subseteq \gamma(\mathcal{S}^\sharp)$. Moreover, if $N \geq 0$ is such that $\mathcal{S}^\sharp_{N+1} = \mathcal{S}^\sharp_N$, then for all $n \geq N$, $\mathcal{S}^\sharp_n = \mathcal{S}^\sharp_N$.

This provides us with an algorithm for automatically computing a sound approximation of \mathcal{S} . It now remains to instantiate the domain Id_2^\sharp and the associated abstract primitives. We will design two abstractions of $\wp(\text{Thr}(S)^* \times \text{Thr}(S)^*)$, each one capturing a particular kind of information.

Our first abstraction captures sequencing information and is based upon an approximation of thread and channel identifiers by *regular languages*. Let $(\text{Reg}, \subseteq, \cup, \emptyset, \cap, \text{Thr}(S)^*)$ be the lattice of regular languages over the alphabet $\text{Thr}(S)$ ordered by set inclusion. We define $\text{Id}_{\text{reg}}^\sharp$ as the product lattice $\text{Reg} \times \text{Reg}$. The concretization γ_{reg} is given by $\gamma_{\text{reg}}(L_1, L_2) = L_1 \times L_2$. The associated abstract primitives are defined as follows:

- $\text{push}_\tau^{\text{reg}}(L_1, L_2) = (L_1 \cdot \tau, L_2)$
- $\text{dpush}_\tau^{\text{reg}}(L_1, L_2) = (L_1 \cdot \tau, L_1 \cdot \tau)$
- $\text{spush}_\tau^{\text{reg}}(L_1, L_2) = (L_1 \cdot \tau, \varepsilon)$
- $\text{swap}^{\text{reg}}(L_1, L_2) = (L_2, L_1)$
- $\text{sync}^{\text{reg}}((L_1, L_2), (L'_1, L'_2)) = \begin{cases} (L_1, L'_1) & \text{if } L_2 \cap L'_2 \neq \emptyset \\ (\emptyset, \emptyset) & \text{otherwise} \end{cases}$

The soundness conditions are easily checked. The element $\varepsilon_2^{\text{reg}}$ is given by $(\varepsilon, \varepsilon)$. Since $\text{Id}_{\text{reg}}^\sharp$ may have infinite strictly increasing chains, we must define a widening operator ∇_2^{reg} . It is sufficient to construct a widening ∇_{reg} on Reg and to apply it componentwise to elements of $\text{Id}_{\text{reg}}^\sharp$. A simple choice for $L_1 \nabla_{\text{reg}} L_2$ consists of quotienting the minimal automaton of $L_1 \cup L_2$ such that any letter of the alphabet may occur at most once in the automaton. The resulting automaton is minimal, and there are finitely many such automata, which ensures the stabilization property.

The second approximation captures counting relations between the components of a tuple of thread or channel identifiers. This will allow us to give nonuniform descriptions of recursively defined communication topologies. Suppose that we are given an infinite set of variables \mathcal{V} . We assign two distinct variables x_τ and y_τ to each element τ of $\text{Thr}(S)$. Now we consider a finite system K of *linear equality constraints* over the variables \mathcal{V} with coefficients in \mathbb{Q} . If we denote

by $|id|_\tau$ the number of occurrences of τ in the sequence id , the concretization $\gamma_{\text{num}}(K)$ of K is the set of all pairs (id, id') such that the following variable assignment:

$$\{x_\tau \mapsto |id|_\tau, y_\tau \mapsto |id'|_\tau \mid \tau \in \text{Thr}(S)\}$$

is a solution of K . The domain of finite systems of linear equality constraints over \mathcal{V} ordered by inclusion of solution sets can be turned into a lattice $\text{Id}_{\text{num}}^\#$. This domain has been originally introduced by Karr [15]. We refer the reader to the original paper for a detailed algorithmic description of lattice operations. We could use more sophisticated domains of computable numerical constraints such as *linear inequalities* [6] or *linear congruences* [12], but the underlying algorithmics is much more involved. Nevertheless, giving a rigorous construction of the abstract primitives on $\text{Id}_{\text{num}}^\#$ would be still very technical. Therefore, for the sake of readability, we only outline the definition of these primitives:

- **push** $_\tau^{\text{num}}(K)$ is the system of constraints K in which we have replaced every occurrence of the variable x_τ by the expression $x_\tau - 1$.
- If K is a system of linear equality constraints, we denote by K_x the system in which we have removed all constraints involving a variable y_τ . Then **dpush** $_\tau^{\text{num}}(K)$ is the system **push** $_\tau^{\text{num}}(K_x)$ with the additional constraints $x_{\tau'} = y_{\tau'}$, for any $\tau' \in \text{Thr}(S)$.
- Similarly, **spush** $_\tau^{\text{num}}(K)$ is the system **push** $_\tau^{\text{num}}(K_x)$ with the additional constraints $y_{\tau'} = 0$, for any $\tau' \in \text{Thr}(S)$.
- **swap** $^{\text{num}}(K)$ is the system in which we have replaced each occurrence of x_τ by y_τ and vice-versa.
- Let K_1 and K_2 be two systems of linear equality constraints. For any $\tau \in \text{Thr}(S)$, let x'_τ and y'_τ be fresh variables of \mathcal{V} . Let K'_2 be the system K_2 in which we have substituted each occurrence of x_τ (resp. y_τ) by x'_τ (resp. y'_τ). We construct the system $K_{1,2}$ as the union of K_1 and K'_2 together with the additional constraints $y_\tau = y'_\tau$, for any $\tau \in \text{Thr}(S)$. Then, we define **sync** $^{\text{num}}(K_1, K_2)$ as the system $K_{1,2}$ in which we have removed all constraints involving a variable y_τ or y'_τ , each remaining variable x'_τ being renamed in y_τ .

Note that a normalization pass (namely a Gauss reduction) has to be performed on the system after or during each of these operations. The element $\varepsilon_2^{\text{num}}$ is given by the system of constraints $\{x_\tau = 0, y_\tau = 0 \mid \tau \in \text{Thr}(S)\}$. Since we only consider systems defined over the finite set of variables $\{x_\tau, y_\tau \mid \tau \in \text{Thr}(S)\}$, we cannot have infinite strictly increasing chains [15]. Therefore, we can use the join operation \sqcup_{num} as a widening.

Example 7. We consider the product of domains $\text{Id}_{\text{reg}}^\#$ and $\text{Id}_{\text{num}}^\#$ and we run the analysis on the system of Example 1 with the notations of Example 2. For the sake of readability, at each step we only write the elements of the abstract configuration that differ from the previous iteration. Moreover, we do not figure trivial constraints of the form $x_\tau = 0$ whenever they can be deduced from the $\text{Id}_{\text{reg}}^\#$ component.

First iteration.

$$\left\{ \begin{array}{l} \langle 0, 1, 1, \text{make}, 0, \text{make} \rangle \mapsto \langle (\varepsilon, \varepsilon), \top_{\text{num}} \rangle, \\ \langle 0, 1, 1, \text{left}_0, 0, \text{left}_0 \rangle \mapsto \langle (\varepsilon, \varepsilon), \top_{\text{num}} \rangle \end{array} \right\}$$

Second iteration.

$$\left\{ \begin{array}{l} \langle 1, 1, 1, \text{mon}, 0, \text{mon} \rangle \mapsto \langle ((0, 1), \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 1, 1, 1, \text{left}_0, 0, \text{left}_0 \rangle \mapsto \langle ((0, 1), \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 1, 1, 1, \text{right}, 1, \text{right} \rangle \mapsto \langle ((0, 1), (0, 1)), x_{(0,1)} = y_{(0,1)} = 1 \rangle, \\ \langle 1, 2, 1, \text{make}, 0, \text{make} \rangle \mapsto \langle ((0, 1), \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 1, 2, 1, \text{right}, 1, \text{right} \rangle \mapsto \langle ((0, 1), (0, 1)), x_{(0,1)} = y_{(0,1)} = 1 \rangle, \\ \langle 2, 1, 1, \text{mon}, 0, \text{mon} \rangle \mapsto \langle ((0, 1), \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 2, 1, 1, \text{left}_0, 0, \text{left}_0 \rangle \mapsto \langle ((0, 1), \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 2, 1, 1, \text{left}, 0, \text{left}_0 \rangle \mapsto \langle ((0, 1), \varepsilon), x_{(0,1)} = 1 \rangle \end{array} \right\}$$

Third iteration.

$$\left\{ \begin{array}{l} \langle 1, 1, 1, \text{mon}, 0, \text{mon} \rangle \mapsto \langle ((0, 1).(\varepsilon + (1, 2)), \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 1, 1, 1, \text{left}_0, 0, \text{right} \rangle \mapsto \left\langle ((0, 1).(1, 2), (0, 1)), \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = 1 \end{array} \right\} \right\rangle, \\ \langle 1, 1, 1, \text{right}, 1, \text{right} \rangle \mapsto \left\langle ((0, 1).(\varepsilon + (1, 2)), (0, 1).(\varepsilon + (1, 2))), \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = y_{(1,2)} \end{array} \right\} \right\rangle, \\ \langle 1, 2, 1, \text{make}, 0, \text{make} \rangle \mapsto \langle ((0, 1).(\varepsilon + (1, 2)), \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 1, 2, 1, \text{right}, 1, \text{right} \rangle \mapsto \left\langle ((0, 1).(\varepsilon + (1, 2)), (0, 1).(\varepsilon + (1, 2))), \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = y_{(1,2)} \end{array} \right\} \right\rangle, \\ \langle 2, 1, 1, \text{mon}, 0, \text{mon} \rangle \mapsto \langle ((0, 1).(\varepsilon + (1, 2)), \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 2, 1, 1, \text{left}_0, 0, \text{left}_0 \rangle \mapsto \langle ((0, 1).(\varepsilon + (1, 2))), \varepsilon \rangle, x_{(0,1)} = 1 \rangle, \\ \langle 2, 1, 1, \text{left}, 0, \text{right} \rangle \mapsto \left\langle ((0, 1).(1, 2), (0, 1)), \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = 1 \end{array} \right\} \right\rangle \end{array} \right\}$$

Fourth iteration.

$$\left\{ \begin{array}{l} \langle 1, 1, 1, \text{mon}, 0, \text{mon} \rangle \mapsto \langle ((0, 1).(1, 2)^*, \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 1, 1, 1, \text{left}_0, 0, \text{right} \rangle \mapsto \left\langle ((0, 1).(1, 2)^*, (0, 1).(\varepsilon + (1, 2))), \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = y_{(1,2)} + 1 \end{array} \right\} \right\rangle, \\ \langle 1, 1, 1, \text{right}, 1, \text{right} \rangle \mapsto \left\langle ((0, 1).(1, 2)^*, (0, 1).(1, 2)^*), \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = y_{(1,2)} \end{array} \right\} \right\rangle, \\ \langle 1, 2, 1, \text{make}, 0, \text{make} \rangle \mapsto \langle ((0, 1).(1, 2)^*, \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 1, 2, 1, \text{right}, 1, \text{right} \rangle \mapsto \left\langle ((0, 1).(1, 2)^*, (0, 1).(1, 2)^*), \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = y_{(1,2)} \end{array} \right\} \right\rangle, \\ \langle 2, 1, 1, \text{mon}, 0, \text{mon} \rangle \mapsto \langle ((0, 1).(1, 2)^*, \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 2, 1, 1, \text{left}_0, 0, \text{left}_0 \rangle \mapsto \langle ((0, 1).(1, 2)^*, \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 2, 1, 1, \text{left}, 0, \text{right} \rangle \mapsto \left\langle ((0, 1).(1, 2)^*, (0, 1).(\varepsilon + (1, 2))), \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = y_{(1,2)} + 1 \end{array} \right\} \right\rangle \end{array} \right\}$$

Fifth iteration.

$$\left\{ \begin{array}{l} \langle 1, 1, 1, left, 0, right \rangle \mapsto \left\langle \begin{array}{l} ((0, 1).(1, 2)^*, (0, 1).(1, 2)^*), \\ \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = y_{(1,2)} + 1 \end{array} \right\} \end{array} \right\rangle, \\ \langle 2, 1, 1, left, 0, right \rangle \mapsto \left\langle \begin{array}{l} ((0, 1).(1, 2)^*, (0, 1).(1, 2)^*), \\ \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = y_{(1,2)} + 1 \end{array} \right\} \end{array} \right\rangle \end{array} \right\}$$

At the sixth iteration step we find the same configuration. Therefore, following Theorem 6 we know that the limit has been reached. Putting all previous computations together, we obtain:

$$\mathcal{S}^\# = \left\{ \begin{array}{l} \langle 0, 1, 1, make, 0, make \rangle \mapsto \langle (\varepsilon, \varepsilon), \top_{\text{num}} \rangle, \\ \langle 0, 1, 1, left_0, 0, left_0 \rangle \mapsto \langle (\varepsilon, \varepsilon), \top_{\text{num}} \rangle, \\ \langle 1, 1, 1, mon, 0, mon \rangle \mapsto \langle ((0, 1).(1, 2)^*, \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 1, 1, 1, left, 0, left_0 \rangle \mapsto \langle ((0, 1), \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 1, 1, 1, left, 0, right \rangle \mapsto \left\langle \begin{array}{l} ((0, 1).(1, 2)^*, (0, 1).(1, 2)^*), \\ \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = y_{(1,2)} + 1 \end{array} \right\} \end{array} \right\rangle, \\ \langle 1, 1, 1, right, 1, right \rangle \mapsto \left\langle \begin{array}{l} ((0, 1).(1, 2)^*, (0, 1).(1, 2)^*), \\ \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = y_{(1,2)} \end{array} \right\} \end{array} \right\rangle, \\ \langle 1, 2, 1, make, 0, make \rangle \mapsto \langle ((0, 1).(1, 2)^*, \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 1, 2, 1, right, 1, right \rangle \mapsto \left\langle \begin{array}{l} ((0, 1).(1, 2)^*, (0, 1).(1, 2)^*), \\ \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = y_{(1,2)} \end{array} \right\} \end{array} \right\rangle, \\ \langle 2, 1, 1, mon, 0, mon \rangle \mapsto \langle ((0, 1).(1, 2)^*, \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 2, 1, 1, left_0, 0, left_0 \rangle \mapsto \langle ((0, 1).(1, 2)^*, \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 2, 1, 1, left, 0, left_0 \rangle \mapsto \langle ((0, 1), \varepsilon), x_{(0,1)} = 1 \rangle, \\ \langle 2, 1, 1, left, 0, right \rangle \mapsto \left\langle \begin{array}{l} ((0, 1).(1, 2)^*, (0, 1).(1, 2)^*), \\ \left\{ \begin{array}{l} x_{(0,1)} = y_{(0,1)} = 1 \\ x_{(1,2)} = y_{(1,2)} + 1 \end{array} \right\} \end{array} \right\rangle \end{array} \right\}$$

This is a very accurate description of the communication topology of the ring. In particular, we are able to distinguish between instances of recursively defined agents and channels. \square

6 Conclusion

We have described a parametric analysis framework for automatically inferring communication topologies of mobile systems specified in the π -calculus. We have instantiated this framework to obtain an effective analysis which is able to give finite descriptions of unbounded communication topologies that distinguish between instances of recursively defined components. To our knowledge this is the only existing analysis of mobile systems (excluding [23]) which can produce

results of that level of accuracy without any strong restriction on the base language. Previous works addressed the issue of communication analysis in CSP [4, 16] or CML [20, 78]. In the latter papers, the analysis techniques heavily rely on CML type information and cannot be applied to more general untyped languages like the π -calculus.

In order to keep the presentation clear within a limited space, we had to make some simplifying assumptions that can be relaxed in many ways, for example by using more expressive abstract domains to denote relations between thread and channel identifiers, like cofibered domains [22, 24], by refining the abstract semantics to take more information into account, like the number of instances of a channel or a thread, or by considering a richer version of the π -calculus with guarded choice, matching and nested replications. Finally, in view of the encodings of classical language constructs (data structures, references, control structures) in the π -calculus, it would be interesting to study the possibility of using a static analysis of the π -calculus as a universal analysis back-end for high-level languages.

Acknowledgements. I wish to thank Radhia Cousot, Patrick Cousot, Ian Mackie and the anonymous referees for useful comments on a first version of this paper.

References

1. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
2. G. Boudol. Asynchrony and the π -calculus. Technical Report 1702, INRIA, 1992.
3. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
4. P. Cousot and R. Cousot. Semantic analysis of communicating sequential processes. In *Seventh International Colloquium on Automata, Languages and Programming*, volume 85 of *LNCS*, 1980.
5. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, August 1992.
6. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Conference on Principles of Programming Languages*. ACM Press, 1978.
7. C. Colby. Analyzing the communication topology of concurrent programs. In *Symposium on Partial Evaluation and Program Manipulation*, 1995.
8. C. Colby. Determining storage properties of sequential and concurrent programs with assignment and structured data. In *Proceedings of the Second International Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 64–81. Springer-Verlag, 1995.
9. P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, 1981.

10. A. Deutsch. A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13. IEEE Computer Society Press, 1992.
11. P. Degano, C. Priami, L. Leth, and B. Thomsen. Analysis of facile programs: A case study. In *Proceedings of the Fifth LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 345–369. Springer-Verlag, 1996.
12. P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT'91*, volume 493. Lecture Notes in Computer Science, 1991.
13. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
14. H.B.M Jonkers. Abstract storage structures. In De Bakker and Van Vliet, editors, *Algorithmic languages*, pages 321–343. IFIP, 1981.
15. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–151, 1976.
16. N. Mercouroff. An algorithm for analysing communicating processes. In *Mathematical Foundations of Programming Semantics*, volume 598 of *LNCS*, 1991.
17. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
18. R. Milner. The polyadic π -calculus: a tutorial. In *Proceedings of the International Summer School on Logic and Algebra of Specification*. Springer-Verlag, 1991.
19. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.
20. H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In 21st *ACM Symposium on Principles of Programming Languages*, 1994.
21. D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, Edinburgh University, 1995.
22. A. Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Proceedings of the Third International Static Analysis Symposium SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 366–382. Springer-Verlag, 1996.
23. A. Venet. Abstract interpretation of the π -calculus. In *Proceedings of the Fifth LOMAPS Workshop on Analysis and Verification of High-Level Concurrent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 51–75. Springer-Verlag, 1996.
24. A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming*, 1999. To appear.

Constructing specific *SOS* semantics for concurrency via abstract interpretation

EXTENDED ABSTRACT

Chiara Bodei¹, Pierpaolo Degano,¹ Corrado Priami²

¹ Dipartimento di Informatica, Università di Pisa
Corso Italia 40, I-56100 Pisa, Italy
{chiara,degano}@di.unipi.it

² Istituto Policattedra, Università di Verona
Ca' Vignal 2, Strada Le Grazie 1, I-37134 Verona, Italy
priami@sci.univr.it

Abstract. Most of the *SOS* semantics for concurrent systems can be derived by abstracting on the inference rules of a concrete transition system, namely the proved transition system. Besides the standard interleaving semantics we mechanically derive the causal transition system for *CCS*, whose definition is particularly difficult and paradigmatic. Its rules are shown to coincide with those presented in the literature. Also, the tree of its computations coincide with that obtained by abstracting the computations of the proved transition system.

Keywords. Concurrency, abstract interpretation, *SOS* semantics, causality, non-interleaving descriptions.

1 Introduction

We apply the theory of abstract interpretation [10] to calculi for concurrency. We carry out our investigation in a pure setting and consider the basic, foundational calculus *CCS* [26]. However, we are confident that more powerful calculi, e.g. those for mobile processes, like the π -calculus [27], require only some technical adjustments to the machinery proposed here. The operational semantics of *CCS* is a transition system, defined following the *SOS* approach. More in detail, a transition for a process P is deduced applying a set of inference rules, driven by the syntax of P . The original transition system describes the evolution of processes in an interleaving style, and a great deal of work has been devoted to define new transition systems expressing also other aspects of process evolutions, both qualitative and quantitative. Among the former, particularly relevant has been the study of transition systems that express the causality between transitions (see [7,14,24,28,6] to mention only a few of a long list of references). Another well-studied non-interleaving qualitative aspect concerns the description of the localities where processes are placed (among the other proposals, see [8,1,2,29]). Quantitative descriptions include transition systems that express temporal aspects [22,30,19,21,16], probabilistic aspects [34,25], and stochastic

ones [20,23,3,9,31]. Besides its interest *in se*, an *SOS* semantics for causality is relevant because it is paradigmatic for others qualitative, non-interleaving description of concurrent systems, as well as for some of the quantitative ones. In particular, many non-interleaving *SOS* definitions have rules in a form similar to those of the causal transition system of [14].

Our starting point is the proved transition system of *CCS* [17], PTS for short. This very concrete representation of systems has been used to describe qualitative and quantitative non-interleaving aspects of concurrent computations [32,33,4]. The transitions are labelled by encodings of their proofs, that encode also most of the aspects briefly mentioned above. The first and the last author used abstract interpretation in [5] to extract the causal computations from the proved computations of the PTS, and sketched how other kinds of computations could be derived, as well. Our main goal here is to push further the use of abstract interpretation in order to mechanically obtain from the PTS the *SOS* inference rules for most of the non-interleaving transition systems mentioned above. We argue that small step operational semantics in *SOS* style for process calculi can be organized in a hierarchy by abstract interpretation, along the lines of abstraction of rule based specifications presented in [11–13].

We work out in full detail the derivation of the causal transition system for *CCS*, possibly the most difficult non-interleaving transition system. We define a (family of) abstraction functions that yield CTS, a variant of the original causal transition system of [14]. As a consequence, the computations of CTS coincide with the abstraction of the computations of PTS, as defined in [5]. The task is easier with Milner’s original transition system [26], that we also derive through abstractions from both the proved and the causal transition systems.

These abstractions constitute a fragment of a hierarchy of transition systems. Our approach is constructive, because the causal *SOS* rules are formally derived by abstraction of the proved ones and the same happens for the interleaving ones. This is a first step towards giving evidence of the power of formal methods over empirical ones.

In the conclusions, we briefly sketch the simple modifications needed to cope with the locality transition system, that we omit here for lack of space. We also give hints on the derivation of temporal transition systems.

2 The concrete domain

We recall the pure *CCS* [26] that we use here as a test-bed. As usual, we denote the countable set of atomic actions by A , the set of co-actions by \overline{A} and the invisible actions with τ . Then $\mathcal{A} = A \cup \overline{A} \cup \{\tau\}$. We also assume a set of agent identifiers with typical element A . Processes (denoted by $P, Q, R, \dots \in \mathcal{P}$) are built from actions and agents according to the syntax

$$P ::= \mathbf{0} \mid \mu.P \mid P + P \mid P|P \mid (\nu a)P \mid A$$

where $\mu \in \mathcal{A}$. Hereafter, we omit the trailing $\mathbf{0}$.

The prefix μ is the first atomic action that the process $\mu.P$ can perform. Summation denotes nondeterministic choice. The operator $|$ describes parallel composition of processes. The restriction operator (νa) prevents P from performing a . A is the invocation of the agent identifier A , that has a unique defining equation of the form $A = P$.

Hereafter, we assume that $(\mathcal{P}/\equiv_P, +, \mathbf{0})$ is a commutative monoid. As in [17], the parallel operator $|$ is *neither* associative *nor* commutative, and this is the only difference with the original CCS definition, apart from the omission of the relabelling operator, which is irrelevant to our present study. The transition system of CCS is in Tab. 1.

$act : \mu.P \xrightarrow{\mu} P$	$ide : \frac{P \xrightarrow{\mu} P'}{A \xrightarrow{\mu} P'}, A = P$
$par_0 : \frac{P \xrightarrow{\mu} P'}{P Q \xrightarrow{\mu} P' Q}$	$sum : \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'}$
$par_1 : \frac{P \xrightarrow{\mu} P'}{Q P \xrightarrow{\mu} Q P'}$	$res : \frac{P \xrightarrow{\mu} P'}{(\nu a)P \xrightarrow{\mu} (\nu a)P'}, \mu \notin \{a, \bar{a}\}$
$com : \frac{P \xrightarrow{\bar{a}} P', Q \xrightarrow{a} Q'}{P Q \xrightarrow{\tau} P' Q'}$	

Table 1. Transition system for CCS .

As anticipated in the introduction, we enrich the labels of the standard interleaving transition system of CCS in the style of [7,15]. It is thus possible to derive different semantic models for CCS by extracting new kinds of labels from the enriched ones (see [17,18]). We start with the definition of the enriched labels and of a function (ℓ) that takes them to the corresponding action labels.

Definition 1. Let $\vartheta \in \{||_0, ||_1\}^*$. Then *proof terms* (with metavariable $\theta \in \Theta$) are defined by the following syntax

$$\theta ::= \vartheta\mu \mid \vartheta\langle ||_0\vartheta_0\mu_0, ||_1\vartheta_1\mu_1 \rangle$$

with $\mu_0 = a$ if and only if $\mu_1 = \bar{a}$, or vice versa.

Function ℓ is defined as $\ell(\vartheta a) = a$ and $\ell(\vartheta\langle ||_0\vartheta_0 a, ||_1\vartheta_1 a' \rangle) = \tau$.

The proved transition system PTS for CCS is in Tab. 2, where the symmetric rule for communication (Com_1) is omitted. These rules are our concrete domain.

$Act : \mu.P \xrightarrow{\mu} P$	$Ide : \frac{P \xrightarrow{\theta} P'}{A \xrightarrow{\theta} P'}, A = P$
$Par_0 : \frac{P \xrightarrow{\theta} P'}{P Q \xrightarrow{\parallel_0 \theta} P' Q}$	$Sum : \frac{P \xrightarrow{\theta} P'}{P + Q \xrightarrow{\theta} P'}$
$Par_1 : \frac{P \xrightarrow{\theta} P'}{Q P \xrightarrow{\parallel_1 \theta} Q P'}$	$Res : \frac{P \xrightarrow{\theta} P'}{(\nu a)P \xrightarrow{\theta} (\nu a)P'}, \ell(\theta) \notin \{a, \bar{a}\}$
$Com_0 : \frac{P \xrightarrow{\vartheta \bar{a}} P', Q \xrightarrow{\vartheta' a} Q'}{P Q \xrightarrow{\langle \parallel_0 \vartheta \bar{a}, \parallel_1 \vartheta' a \rangle} P' Q'}$	

Table 2. Proved transition system for *CCS*.

The proved transition system differs from the one in Tab. 1 in the rules for parallel composition and communication. Rule Par_0 (Par_1) adds to the label a tag \parallel_0 (\parallel_1) to record that the left (right) component is moving. The rule Com_0 has in its conclusion a pair instead of a τ to record the components which interacted.

The standard interleaving semantics [26] is obtained from the proved one by relabelling each transition through function ℓ in Def. 1. The relabelling is an abstraction from the proved transition system to the interleaving transition system semantics. This result is made precise in Subsection 6.2.

3 The abstract domain

Causal *CCS* (*CCCS*) is a version of *CCS* introduced in [14] to give full account to causality relations between transitions. Processes are extended with a family of operators $K ::$, with $K \in \wp_{fin}(\mathbb{N})$. The intended meaning of K in $K :: P$ is to encode the causes of the process P , i.e. the transition occurrences enabling P .

The *CCCS* processes, called *extended processes* and denoted by $t, t', \dots \in \mathcal{T}$, are defined according to the syntax

$$t ::= K :: P \mid t|t \mid (\nu a)t.$$

We assume that the operators $K ::$ distribute over the restriction (νa) and the parallel operator $|$, i.e. we assume on extended processes the least congruence \equiv that satisfies the following clauses

$$\bullet K :: (\nu r)t \equiv (\nu r)K :: t \quad \bullet K :: (t|t') \equiv (K :: t)|(K :: t')$$

Also we let $K :: K' :: P = (K \cup K') :: P$; and we write $\mathcal{K}(t)$ for the set of causes in t , k for the singleton $\{k\}$, and $K\{K'/k\}$ for $(K \setminus k) \cup K'$. Moreover, we let $(K :: P)\{K'/k\} = K\{K'/k\} :: P$ and we homomorphically extend this replacement of a set of causes K for a single cause k onto extended processes t .

Causal transitions have the form $t \xrightarrow{K, \mu}_{K, k} t'$, where μ is the standard action, $k \in \mathcal{N}$ is the unique name of the transition and K represents the set of the transitions that enable k . The causal transition system in Tab. 3 is a variant of the original one of [14] (see also [24]), where the function f is¹

$$f(K, \mu) = \begin{cases} K; k & \text{if } \mu \neq \tau \\ \emptyset; \emptyset & \text{otherwise} \end{cases}.$$

We write $\pi_2(f(K, \mu))$ for the second component of $f(K, \mu)$ and, by abuse of notation, $\pi_2(f(K, \mu)) \notin K$, assuming it true if $\pi_2(f(K, \mu)) = \emptyset$. As usual, we omit in Tab. 3 the rule $Kcom_1$, symmetric to $Kcom_0$.

$Kact : K :: \mu.P \xrightarrow{f(K, \mu)} \pi_2(f(K, \mu)) \cup K :: P, \pi_2(f(K, \mu)) \notin K$	
$Kide : \frac{t \xrightarrow{\mu}_{f(K, \mu)} t'}{t'' \xrightarrow{\mu}_{f(K, \mu)} t'}, t'' = t$	$Kres : \frac{t \xrightarrow{\mu}_{f(K, \mu)} t'}{(\nu a)t \xrightarrow{\mu}_{f(K, \mu)} (\nu a)t'}, \mu \notin \{a, \bar{a}\}$
$Kpar_0 : \frac{t \xrightarrow{\mu}_{f(K, \mu)} t'}{t t'' \xrightarrow{\mu}_{f(K, \mu)} t' t''}, \pi_2(f(K, \mu)) \notin \mathcal{K}(t'')$	$Ksum : \frac{t \xrightarrow{\mu}_{f(K, \mu)} t'}{t + t'' \xrightarrow{\mu}_{f(K, \mu)} t'}$
$Kpar_1 : \frac{t \xrightarrow{\mu}_{f(K, \mu)} t'}{t'' t \xrightarrow{\mu}_{f(K, \mu)} t'' t'}, \pi_2(f(K, \mu)) \notin \mathcal{K}(t')$	$Kcom_0 : \frac{t_0 \xrightarrow{a}_{K_0; k} t'_0, t_1 \xrightarrow{\bar{a}}_{K_1; k} t'_1}{t_0 t_1 \xrightarrow{\tau}_{\emptyset; \emptyset} t'_0\{K_1/k\} t'_1\{K_0/k\}}$

Table 3. Causal transition system for *CCCS*.

4 Rule-based inductive definitions

We first recall some definitions and results from [11]. Then, we instantiate them in our setting.

¹ This function is introduced to avoid duplicating the rules: one set to handle τ -transitions and the other for the visible transitions, as done in [24].

Definition 2. An *inductive definition* is a quadruple $\mathcal{F} = \langle U, \Phi, \perp, \sqcup \rangle$ such that:

- the *universe* U is a set;
- Φ is the set of *rule instances* $\frac{P}{c}$, with $P \subseteq U$ and $c \in U$;
- $\perp \in U$ is the *basis*;
- $\sqcup \in \wp(\wp(U)) \rightarrow \wp(U)$, is the (partial) *join operator* and the *induced ordering* \sqsubseteq is $x \sqsubseteq y$ iff $x \sqcup y = y$ and is a partial order on $\wp(U)$.

Definition 3.

The *operator* $\overline{\Phi}$ induced by $\mathcal{F} = \langle U, \Phi, \perp, \sqcup \rangle$ is $\overline{\Phi} : \wp(U) \rightarrow \wp(U)$ such that

$$\overline{\Phi}(X) = \{c \in U \mid \exists P \subseteq X : \frac{P}{c} \in \Phi\}$$

The inductive definition \mathcal{F} is *monotonic* whenever $\overline{\Phi}$ is monotonic and $\perp \sqsubseteq \overline{\Phi}(\perp)$. Moreover \mathcal{F} is *on a cpo* (resp. on a complete lattice) iff $\langle \wp(U), \sqsubseteq, \perp, \sqcup \rangle$ (resp. $\langle \wp(U), \sqsubseteq, \perp, \sqcup, \sqcap, \sqcup \rangle$) is a cpo (resp. a complete lattice).

Definition 4. Given $\mathcal{F} = \langle U, \Phi, \perp, \sqcup \rangle$ and $\mathcal{F}' = \langle U', \Phi', \perp', \sqcup' \rangle$, we write $\mathcal{F} \xleftrightarrow[\gamma]{\alpha} \mathcal{F}'$ for $\langle \wp(U), \sqsubseteq, \perp, \sqcup \rangle \xleftrightarrow[\gamma]{\alpha} \langle \wp(U'), \sqsubseteq', \perp', \sqcup' \rangle$, whenever $\langle \alpha, \gamma \rangle$ is a Galois connection, i.e. if and only if

- $\langle \wp(U), \sqsubseteq \rangle$ and $\langle \wp(U'), \sqsubseteq' \rangle$ are posets;
- $\alpha \in \wp(U) \rightarrow \wp(U')$ and $\gamma \in \wp(U') \rightarrow \wp(U)$;
- $\forall x \in \wp(U), \forall y \in \wp(U') : [\alpha(x) \sqsubseteq' y] \text{ if and only if } [x \sqsubseteq \gamma(y)]$.

The following proposition (corresponding to Prop.53 in [11]), will be useful later.

Proposition 5. Let $\mathcal{F} = \langle U, \Phi, \perp, \sqcup \rangle$ be such that

1. \mathcal{F} is a monotonic inductive definition on the cpo $\langle \wp(U), \sqsubseteq, \perp, \sqcup \rangle$,
2. $\langle \wp(U'), \sqsubseteq', \perp' \rangle$ be a partial order,
3. $\alpha \in \wp(U) \rightarrow \wp(U')$ be a complete \sqcup - and \sqcup -morphism such that $\forall \frac{P}{c} \in \Phi$,
 $\forall X \subseteq U : \alpha(P) \sqsubseteq \alpha(X) \text{ implies } \exists \frac{P'}{c'} \in \Phi' : P' \subseteq X \text{ and } \alpha(\{c\}) \sqsubseteq \alpha(\{c'\})$.

Define $\Phi' = \{ \frac{\alpha(P)}{c'} \mid \frac{P}{c} \in \Phi, \text{ with } c' \in \alpha(c) \}$ and $\perp = \alpha(\perp)$.

Then $\mathcal{F}' = \langle U', \Phi', \perp', \sqcup' \rangle$ is a well formed inductive definition (i.e. a monotonic or extensive definition on a cpo) such that $\alpha : \mathcal{F} \rightarrow \mathcal{F}'$.

If moreover $\langle \wp(U), \sqsubseteq, \perp, \sqcup, \sqcap \rangle$ is a complete lattice then $\mathcal{F} \xleftrightarrow[\gamma]{\alpha} \mathcal{F}'$, where $\gamma \in \wp(U') \rightarrow \wp(U)$ is $\gamma(Y) = \sqcup \{X \in U \mid \alpha(X) \sqsubseteq' Y\}$.

More precisely, under the conditions of the above proposition we have the following (for the notation used and more technical details see Def. 46 and Proposition 53 in [11]). The abstraction function α preserves least fixed points lfp , because $\mathcal{F} \xleftrightarrow[\gamma]{\alpha} \mathcal{F}'$ implies that $\alpha(lfp_{\sqsubseteq}(\overline{\Phi})) = lfp_{\sqsubseteq'}(\overline{\Phi'})$.

In the following we use a special form of inductive definition only, namely the positive one. These definitions are well-formed, hence they have always the fixed point $\overline{\Phi}_U^\infty(\emptyset)$.

Definition 6. The inductive definition $\mathcal{F}_P = \langle U_P, \Phi_P, \perp, \sqcup \rangle$ is *positive*, and is written as $\langle U, \Phi \rangle$, whenever \perp is the emptyset \emptyset and \sqcup is the set union \cup .

The set of all the possible transitions involving *CCS* processes can be generated by a rule-based inductive definition. In this way, instead of reasoning about the classical schemata of rules and axioms, we consider a set Φ of rule instances, where axioms have empty premises. Some of the rules in Tables 2 and 3 have side-conditions, that can also be seen as additional premises. In fact, each rule with a side-condition stands for a set of rules, therefore we can safely use a rule-based inductive definition.

4.1 The concrete domain \mathcal{F}_P

Here we give a rule-based inductive definition of the proved transitions of *CCS*.

Definition 7. The inductive definition $\mathcal{F}_P = \langle U_P, \Phi_P \rangle$ is the positive inductive definition, where:

- $U_P = \{P \xrightarrow{\theta} P' \mid P, P' \in \mathcal{P}, \theta \in \Theta\}$ is the universe, and
- $\Phi_P = \{ \frac{P}{c} \mid P \subseteq U_P, c \in U_P \text{ and } \frac{P}{c} \text{ is an instance of } \text{Act, Ide, Res, Sum, Par}_0, \text{Par}_1, \text{Com}_0, \text{Com}_1 \}$ is the set of rule instances.

The following properties of \mathcal{F}_P are easy to establish.

Proposition 8. *The positive inductive definition \mathcal{F}_P is monotonic and it is well-formed.*

Proof. We only need to prove that the operator $\overline{\Phi}_P$ is monotonic and that $\emptyset \subseteq \overline{\Phi}_P(\emptyset)$. Let $X \subseteq Y \subseteq U_P$. Since $c \in \overline{\Phi}_P(X)$ implies $\exists P \subseteq X : \frac{P}{c} \in \Phi_P$ and $X \subseteq Y$, then $c \in \overline{\Phi}_P(Y)$. Finally, $\emptyset \subseteq \overline{\Phi}_P(\emptyset)$ holds because $\overline{\Phi}_P(\emptyset)$ is a set.

Proposition 9. $\langle \wp(U_P), \subseteq, \emptyset, U_P, \cup, \cap \rangle$ is a complete lattice.

4.2 The abstract domain \mathcal{F}_C

The set of all the possible causal transitions involving *CCCS* processes can be generated by a rule-based positive inductive definition, as well.

Definition 10. $\mathcal{F}_C = \langle U_C, \Phi_C \rangle$ is the positive inductive definition, where:

- $U_C = \{t \xrightarrow[\mu]{f(K, \mu)} t' \mid t, t' \in \mathcal{T}, \mu \in \text{Act}, K \in \wp_{fin}(\mathbb{N})\}$ is the universe;
- $\Phi_C = \{ \frac{P}{c} \mid P \subseteq U_C, c \in U_C \text{ and } \frac{P}{c} \text{ is an instance of } K\text{act}, K\text{ide}, K\text{res}, K\text{sum}, K\text{par}_0, K\text{par}_1, K\text{com}_0, K\text{com}_1 \}$ is the set of rule instances.

Just as for \mathcal{F}_P , we have the following.

Proposition 11. *The positive inductive definition \mathcal{F}_C is monotonic and well-formed.*

Proposition 12. $\langle \wp(U_C), \subseteq, \emptyset, U_C, \cup, \cap \rangle$ is a complete lattice.

5 The abstract interpretation function

To abstract inductive definitions, we use an abstraction of subsets of the universe, according to Proposition 5. Our abstraction is based on a family of abstraction functions, which act on the components of the rule schemata and which are parameterized on a class S defined according to the syntax

$$S ::= K \mid (S|S).$$

We write S^i for the i^{th} (from the left) set $K \in \wp_{fin}(N)$ of (the frontier of the tree) S . For instance if $S = (K|(K'|K''))|(H|H')$, then $S^4 = H$.

The family of functions α_S is such that

$$\bullet \alpha_K(P) = K :: P \qquad \bullet \alpha_{S_0|S_1}(P \mid Q) = \alpha_{S_0}(P) \mid \alpha_{S_1}(Q).$$

Moreover we assume that

$$\begin{aligned} - K :: \alpha_H(P) &= \alpha_{K \cup H}(P); \\ - K :: \alpha_{S_0|S_1}(P \mid Q) &= K :: \alpha_{S_0}(P) \mid K :: \alpha_{S_1}(Q); \\ - (S_0 \mid S_1)\{H/k\} &= S_0\{H/k\} \mid S_1\{H/k\}. \end{aligned}$$

As an example of how an abstraction function works, consider the S introduced above and the process $R = (P|(P'|P''))|(a.Q|Q')$. Then

$$\begin{aligned} \alpha_S(R) &= \alpha_{(K|(K'|K''))}(P|(P'|P'')) \mid \alpha_{(H|H')}(a.Q|Q') = \\ &= \alpha_K(P) \mid (\alpha_{K'}(P') \mid \alpha_{K''}(P'')) \mid (\alpha_H(a.Q) \mid \alpha_{H'}(Q')) = \\ &= (K :: P \mid (K' :: P' \mid K'' :: P'')) \mid (H :: a.Q \mid H' :: Q') = t_R. \end{aligned}$$

Now, we can define the abstraction function α_c^p that maps proved transitions in causal transitions. It will be used to obtain the causal transition system by abstracting the rules of the proved one.

Definition 13. Let $\alpha_c^p : \wp(U_P) \rightarrow \wp(U_C)$ be defined as

$$\begin{aligned} \alpha_c^p(X) &= \{\alpha_c^p(P \xrightarrow{\theta\tau} P') = \alpha_S(P) \xrightarrow[\emptyset, \emptyset]{\tau} \alpha_S(P') \mid P \xrightarrow{\vartheta\tau} P' \in X\} \cup \\ &\quad \{\alpha_c^p(P \xrightarrow{\theta(\vartheta a, \vartheta' a')} P') = \alpha_{S_0}(P) \xrightarrow[\emptyset, \emptyset]{\tau} \alpha_{S_1}(P') \mid P \xrightarrow{\vartheta(\vartheta' a', \vartheta'' a'')} P' \in X, \\ &\quad \exists i, j : S_1^i = S_1^j = S_0^i \cup S_0^j \text{ and } \forall p \neq i, j : S_1^p = S_0^p\} \cup \\ &\quad \{\alpha_c^p(P \xrightarrow{\vartheta a} P') = \alpha_{S_0}(P) \xrightarrow[\vec{K}, \vec{k}]{a} \alpha_{S_1}(P') \mid P \xrightarrow{\vartheta a} P' \in X, \forall p : k \notin S_0^p \text{ and} \\ &\quad \exists i, K : S_0^i = K, S_1^i = K \cup \{k\}, \forall j \neq i : S_1^j = S_0^j\} \end{aligned}$$

To see our definition at work, consider again the α_S and R introduced above, and the following proved transition $R \xrightarrow{\parallel_1 \parallel_0^a} R' = (P|(P'|P''))|(Q|Q')$. Then, its abstraction is $t_R \xrightarrow[\vec{H}, \vec{h}]{a} t_{R'}$, for $h \notin K \cup K' \cup K'' \cup H \cup H'$ and

$$S' = (K|(K'|K''))|(H \cup \{h\}|H'), \text{ where}$$

$$t_{R'} = \alpha_{S'}(R') = (K :: P \mid (K :: P' \mid K'' :: P'')) \mid (H \cup \{h\} :: Q \mid H' :: Q').$$

It is easy to deduce this causal transition with the rules in Tab. 3. The interested reader may wish to abstract step by step its whole proved derivation. We wish to

$$\begin{aligned}
\alpha_r(Act) &= \alpha_K(\mu.P) \xrightarrow{f(K, \mu)} \pi_2(f(K, \mu)) :: \alpha_K(P), \quad \pi_2(f(K, \mu)) \notin K \\
\alpha_r(De) &= \frac{\alpha_S(P) \xrightarrow{f(K, \ell(\theta))} \pi_2(f(K, \ell(\theta))) :: \alpha_{S'}(P')}{\alpha_S(A) \xrightarrow{f(K, \ell(\theta))} \alpha_{S'}(P')}, \quad \alpha_S(A) = \alpha_S(P) \\
\alpha_r(Res) &= \frac{\alpha_S(P) \xrightarrow{f(K, \ell(\theta))} \alpha_{S'}(P')}{\alpha_S(\nu a)P \xrightarrow{f(K, \ell(\theta))} \alpha_{S'}((\nu a)P')}, \quad \ell(\theta) \notin \{a, \bar{a}\} \\
\alpha_r(Sum) &= \frac{\alpha_S(P) \xrightarrow{f(K, \ell(\theta))} \alpha_{S'}(P')}{\alpha_S(P + Q) \xrightarrow{f(K, \ell(\theta))} \alpha_{S'}(P')} \\
\alpha_r(Par_0) &= \frac{\alpha_{S_0}(P) \xrightarrow{f(K, \ell(\theta))} \alpha_{S'_0}(P')}{\alpha_{S_0|S_1}(P \mid Q) \xrightarrow{f(K, \ell(\theta))} \alpha_{S'_0|S_1}(P' \mid Q)}, \quad \forall i : \pi_2(f(K, \ell(\theta))) \notin S_i^i \\
\alpha_r(Par_1) &= \frac{\alpha_{S_1}(Q) \xrightarrow{f(K, \ell(\theta))} \alpha_{S'_1}(Q')}{\alpha_{S_0|S_1}(P \mid Q) \xrightarrow{f(K, \ell(\theta))} \alpha_{S_0|S'_1}(P \mid Q')}, \quad \forall i : \pi_2(f(K, \ell(\theta))) \notin S_0^i \\
\alpha_r(Com_0) &= \frac{\alpha_{S_0}(P_0) \xrightarrow{f(K_0; a)} \alpha_{S'_0}(P'_0), \alpha_{S_1}(P_1) \xrightarrow{f(K_0; \bar{a})} \alpha_{S'_1}(P'_1)}{\alpha_{S_0|S_1}(P_0 \mid P_1) \xrightarrow{\tau_{\emptyset; \emptyset}} \alpha_{S'_0\{K_1/k\} \mid S'_1\{K_0/k\}}(P'_0 \mid P'_1)}
\end{aligned}$$

Table 4. The abstraction interpretation α_r of rule instances.

use Proposition 5 to establish a Galois connection between the definitions of the proved and the causal transition systems. We already know that the abstract domain is a complete lattice (Proposition 12). We are left to show that our abstraction function α_c^p enjoys the properties required by item 3 of Proposition 5.

Proposition 14. *The function α_c^p a complete \cup -morphism,*

$$i.e. \forall X \subseteq U_P : \cup X \in U_P \text{ implies } \cup \alpha_c^p(X) = \alpha_c^p(\cup X).$$

Proof. It is easy to prove that the function α_c^p is monotone. We now prove the double inclusion \subseteq and \supseteq (note that induced ordering is the same for both domains).

We start with \subseteq . By definition of \cup , $\forall X \subseteq U_P$, $\alpha_c^p(X) \subseteq \cup \alpha_c^p(X)$ and $X \subseteq \cup X$. Then, by monotonicity of α_c^p , it is $\alpha_c^p(X) \subseteq \alpha_c^p(\cup X)$. Since \cup is the *least* upper bound, $\cup' \alpha_c^p(X) \subseteq \alpha_c^p(\cup X)$.

We now prove \supseteq . Let FK be the function introduced in Def. 25 that deletes the causes from an extended process, restoring a pure *CCS* process.

By hypothesis $\cup \alpha_c^p(X) = \{t_i \xrightarrow{\mu_i}_{K_i; k_i} t'_i \mid i \in I\} \cup \{t_j \xrightarrow{\tau}_{\emptyset; \emptyset} t'_j \mid j \in J\} \subseteq U_C$. Then,

by construction, there exists a set $Y = \{P_p \xrightarrow{\theta_p} P'_p \mid FK(t_p) = P_p, FK(t'_p) = P'_p, \ell(\theta_p) = \mu_p \mid p \in I \cup J\}$, such that $\alpha_c^p(Y) = \cup \alpha_c^p(X)$. First we prove $\forall \tilde{Y} \subseteq X : \tilde{Y} \subseteq Y$. Per absurdum, let $S \subseteq X$ such that $S \supset Y$. Then, by monotonicity of α_c^p , $\alpha_c^p(S) \supseteq \alpha_c^p(Y) = \cup \alpha_c^p(X)$, against the hypothesis of $\cup \alpha_c^p(X)$ to be the least upper bound. Then, $\forall \tilde{Y} \subseteq X : \tilde{Y} \subseteq \cup X$, as well. From this and from $\tilde{Y} \subseteq Y$, $\cup X \subseteq Y$, because $\cup X$ is the *least* upper bound. By monotonicity of α_c^p , we conclude $\alpha_c^p(\cup X) \subseteq \alpha_c^p(Y) = \cup \alpha_c^p(X)$.

Proposition 15. *The function α_c^p is such that $\forall \frac{P}{c} \in \Phi_P$, $\forall X \subseteq U_P$:*

$\alpha_c^p(P) \subseteq \alpha_c^p(X)$ *implies* $\exists \frac{P'}{c'} \in \Phi_P : P' \subseteq X$ *and* $\alpha_c^p(\{c\}) \subseteq \alpha_c^p(\{c'\})$.

Proof. Since $\alpha_c^p(P) \neq \alpha_c^p(P')$, whenever $P \neq P'$, $\alpha_c^p(P) \subseteq \alpha_c^p(X)$ implies $P \subseteq X$. Hence, we only need to put $P = P'$.

To define the abstract interpretation of the operator Φ , we introduce in Tab. 4 the auxiliary family of functions α_r . Note that $\alpha_r(\frac{P}{c}) = \frac{\alpha_c^p(P)}{c'}$ with $c' \in \alpha_c^p(c)$ (cf. the definition of Φ' in Proposition 5). We can now establish the required Galois connection, that preserves least fixed points.

Theorem 16. *Let $\Phi' = \{\alpha_r(\frac{P}{c}) \mid \frac{P}{c} \in \Phi_P\}$ and $\emptyset = \alpha_r(\emptyset)$. The positive inductive definition $\mathcal{F}' = \{\alpha_c^p(U), \Phi'\}$ is well formed and such that $\mathcal{F}_P \xrightarrow[\gamma_P^c]{\alpha_P^c[=]} \mathcal{F}'$, i.e. $\langle \alpha_c^p, \gamma_P^c \rangle$ is a Galois connection, where $\gamma_P^c : \wp(U_C) \rightarrow \wp(U_P)$ is*

$$\gamma_P^c(Y) = \begin{cases} \cup \{X \in U_P \mid \alpha_c^p(X) \subseteq Y\} & \text{if } Y \in \alpha_c^p(U_P) \\ U_P & \text{if } Y \notin \alpha_c^p(U_P) \end{cases}$$

Proof. By Proposition 5, whose hypotheses are satisfied because of Propositions 8, 9, 11, 12, 14, 15.

Although $\alpha_c^p(U_P) \subset U_C$, we have established exactly what we would like to obtain. In fact, it is immediate to prove the following property, because the rules in Tables 3 and 4 can be easily put in bijection.

Proposition 17. $\alpha_r(\Phi_P) = \Phi_C$.

The above correspondence suffices to establish that the computations of the causal transition system obtained via abstraction α_c^p coincide with the abstraction of the proved computations as defined in [5]. Actually, a stronger result holds: the trees of computations coincide, so the non deterministic aspects of computations are preserved, as well. Lack of space prevents us from recalling the relevant definitions and results of [5]. We simply introduce some notation.

Let $Ptree(P)$ be the tree defined as the set of the proved computations starting from P , ordered by prefix. Similarly, the tree of causal computations is $Ktree(P)$, with transitions derived by the rules in Tab. 3; and finally let $\alpha_c^p Ptree(P)$ be the tree of computations with transitions deduced according to $\alpha_c^p(\Phi_P)$. We denote by $\langle \bar{\alpha}, \bar{\gamma} \rangle$ the Galois connection of Def. 17 in [5], where $\bar{\alpha}$ maps the tree of proved computations $Ptree(P)$ to the tree of causal computations $Ktree(P)$. Now, we can state the following.

Theorem 18. *For all $P \in \mathcal{P}$, we have that $\alpha_c^p Ptree(P) = Ktree(P)$.*

Proof. The bijection between Tab. 3 and Tab. 4 suffices, because $\bar{\alpha}(Ptree(P)) = Ktree(P)$ by Theorem 18 of [5].

6 A fragment of a hierarchy

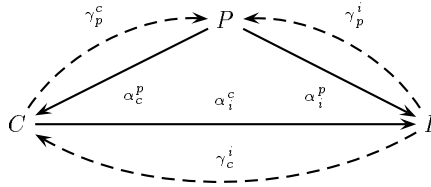


Fig. 1. The hierarchy of semantics.

We now define a simple hierarchy of transition systems involving the proved, the causal and the standard interleaving semantics of CCS . The hierarchy in Fig. 1 is established by studying three abstraction functions: from the proved to the causal model and to the interleaving model, and from the causal to the interleaving one. This fragment of a hierarchy is formally constructed, and in the conclusions we sketch how other models can be derived by further constructive abstractions of the proved model.

6.1 The abstract domain \mathcal{F}_I

We start with the definition of the abstract domain made of the standard interleaving transition system of CCS [26]. The set of all the possible transitions involving CCS processes can be generated by the following positive rule-based inductive definition.

Definition 19. $\mathcal{F}_I = \langle U_I, \Phi_I \rangle$ is the positive inductive definition, where:

- $U_I = \{P \xrightarrow{\mu} P' \mid P, P' \in \mathcal{P}, \mu \in Act\}$ is the universe;
- $\Phi_I = \{\frac{P}{c} \mid P \subseteq U_I, c \in U_I \text{ and } \frac{P}{c} \text{ is an instance of } act, ide, res, sum, par_0, par_1, com_0, com_1\}$ is the set of rule instances.

Just as for \mathcal{F}_C , we have the following.

Proposition 20. *The positive inductive definition \mathcal{F}_I is monotonic and well-formed.*

Proposition 21. $\langle \wp(U_I), \subseteq, \emptyset, U_I, \cup, \cap \rangle$ is a complete lattice.

6.2 The abstraction from proved to interleaving transition systems

We make now precise the observation made at the end of Section 2, by defining a Galois connection between \mathcal{F}_P and \mathcal{F}_I . So one can easily recover the standard interleaving transition system from the proved one.

Definition 22. Let $\alpha_i^p : \wp(U_P) \rightarrow \wp(U_I)$ be defined as

$$\alpha_i^p(X) = \{\alpha_i^p(P \xrightarrow{\theta} P') = P \xrightarrow{\ell(\theta)} P' \mid P \xrightarrow{\theta} P' \in X\}, \text{ where } \ell \text{ is as in Def. 1.}$$

Also in this case is possible to reformulate the results relative to the Galois connection and the consequent correspondences. Note again that $\alpha_r''(\frac{P}{c}) = \frac{\alpha_i^p(P)}{c'}$ with $c' \in \alpha_i^p(c)$.

Theorem 23. *Let $\Phi'' = \{\alpha_r''(\frac{P}{c}) \mid \frac{P}{c} \in \Phi_P\}$ and $\emptyset = \alpha_r''(\emptyset)$. The positive inductive definition $\mathcal{F}'' = \{\alpha_i^p(U_P), \Phi''\}$ is well formed and is such that*

$\mathcal{F}_P \xleftrightarrow[\gamma_p^i]{\alpha_i^p[=]} \mathcal{F}'_I$, i.e. $\langle \alpha_i^p, \gamma_p^i \rangle$ is a Galois connection, where $\gamma_p^i : \wp(U_I) \rightarrow \wp(U_P)$ is

$$\gamma_p^i(Y) = \begin{cases} \cup \{X \in U_P \mid \alpha_i^p(X) \subseteq Y\} & \text{if } Y \in \alpha_i^p(U_P) \\ U_P & \text{if } Y \notin \alpha_i^p(U_P) \end{cases}$$

The following proposition says that the induced operator Φ'' above is indeed Φ_I , and justifies the use of $\xrightarrow{\mu}$ in Def. 22.

Proposition 24. $\alpha_r''(\Phi_P) = \Phi_I$.

6.3 The abstraction from causal to interleaving transition systems

Our next abstraction is the function α_i^c that maps causal transitions in standard interleaving transitions. It will be used to obtain the interleaving transition system by abstracting the rules of the causal one. The steps are the same as above.

Definition 25. Let $\alpha_i^c : \wp(U_C) \rightarrow \wp(U_I)$ be defined as

$$\alpha_i^c(X) = \{\alpha_i^c(t \xrightarrow[\mu]{f(K)} t') = FK(t) \xrightarrow{\mu} FK(t') \mid t \xrightarrow[\mu]{f(K)} t' \in X\}.$$

where FK deletes the causes from extended processes, defined as

$$FK(K :: t) = FK(t), FK(t|t') = FK(t)|FK(t'), FK((\nu a)t) = (\nu a)FK(t).$$

Theorem 26. Let $\Phi''' = \{\alpha_r'''(\frac{P}{c}) \mid \frac{P}{c} \in \Phi_C\}$ and $\emptyset = \alpha_r'''(\emptyset)$. The positive inductive definition $\mathcal{F}''' = \{\alpha_i^c(U_C), \Phi'''\}$ is well formed and is such that

$\mathcal{F}_C \xrightarrow[\gamma_c^i]{\alpha_i^c[=]} \mathcal{F}'''_I$, i.e. $\langle \alpha_i^c, \gamma_c^i \rangle$ is a Galois connection, where $\gamma_c^i : \wp(U_I) \rightarrow \wp(U_C)$ is

$$\gamma_c^i(Y) = \begin{cases} \cup \{X \in U_C \mid \alpha_i^c(X) \subseteq Y\} & \text{if } Y \in \alpha_i^c(U_C) \\ U_C & \text{if } Y \notin \alpha_i^c(U_C) \end{cases}$$

Note that $FK(t)$ is a standard *CCS* process and the following proposition justifies the presence of $\xrightarrow{\mu}$ in Def. 25.

Proposition 27. $\alpha_r'''(\Phi_C) = \Phi_I$.

6.4 The hierarchy

The commutativity of the diagram in Fig. 1 is now immediate.

Theorem 28. $\alpha_i^c(\alpha_c^p(\Phi_P)) = \alpha_i^p(\Phi_P) = \Phi_I$.

The above theorem suffices to show a correspondence between the trees of computations in the various models. The trees of computations obtained by abstracting proved computations to interleaving ones coincide with the trees of the computations based on the standard interleaving semantics. The same happens abstracting causal computations. Therefore, the non deterministic aspects of agent behaviour are fully respected. This supports our claim that formal constructive methods can be used to mechanically derive, clarify and compare various different models for concurrency.

Let $Stree(P)$ be the synchronization tree defined as the set of the interleaving *CCS* computations starting from P , ordered by prefix [26]. Also, let $\alpha_i^p Ptree(P)$ and $\alpha_i^c Ktree(P)$ be the trees of computations with transitions deduced with $\alpha_i^p(\Phi_P)$ and with $\alpha_i^c(\Phi_C)$. The following is immediate.

Theorem 29. For all $P \in \mathcal{P}$, $\alpha_i^p Ptree(P) = Stree(P) = \alpha_i^c Ktree(P)$.

7 Conclusions

We have constructed the causal and the interleaving transition systems for *CCS* by abstracting on the rules of the proved transition system. Lack of space prevents us from giving more evidence that quite similar constructions are possible, and give most of the interleaving and the non-interleaving *SOS* definitions

presented in the literature for concurrency calculi. Their designers aimed at describing different qualitative and quantitative aspects of process evolutions, and a considerable effort has been spent in their actual *SOS* definitions.

Our proposal may help in mechanically deriving these *SOS* definitions, keeping in mind only the effect that a *single* transition has on the state of the whole system of concurrent processes. Consider for instance the effect that a visible action a has on the α_S in Def. 13: only the causes K of the process performing a are affected — this obvious fact, together with exchange of causes in communications, are sufficient to obtain the rules for the causal transition system.

Despite space limitation, we would like to give some hints of three more abstraction functions, thus showing that our construction is in a sense paradigmatic.

Consider the location transition system of [8]. It was designed to keep track of the sites (taken from a given set *Location*) where the visible actions actually occur, and it assumes that the sites of invisible ones cannot be detected. Having this in mind, it is easy to define the location abstraction functions α_l^p , mimicking Def. 13, as follows.

$$\begin{aligned} \alpha_l^p(X) = & \{ \alpha_l^p(P \xrightarrow{\theta} P') = \alpha_L(P) \xrightarrow{\tau} \alpha_L(P') \mid P \xrightarrow{\theta} P' \in X, \ell(\theta) = \tau \} \cup \\ & \{ \alpha_l^p(P \xrightarrow{\vartheta a} P') = \alpha_L(P) \xrightarrow{a}_{lu} \alpha_{L'}(P') \mid P \xrightarrow{\vartheta a} P' \in X, \\ & \forall p : l \notin L_0^p \text{ and } \exists i : L_0^i = u, L_1^i = lu, \text{ and } \forall j \neq k : L_1^j = L_0^j \}, \end{aligned}$$

where $l \in \text{Location}$, $u \in \text{Location}^*$ and L is either u or $L|L$ (i.e. L is a tree of strings of locations, rather than a tree of sets of natural numbers), and α_L is a family of abstraction functions defined just as α_S .

Indeed, the only difference between Def. 13 and the one sketched above is that communications do not exchange causes, a typical feature of causality, not shared by location models. It is easy then to establish a Galois connection $\langle \alpha_l^p, \gamma_p^l \rangle$ and results similar to Theorems 16 and 18.

At this point, the reader familiar with the local/global transition system of [24] can easily obtain it by defining an abstraction function $\alpha_{l/g}^p$ that combines α_c^p and α_l^p and that enjoys all the properties these abstraction functions have.

As a third example, we consider a *SOS* semantics that expresses quantitative aspects of concurrent computations. Consider the timed transition system of [19], whose transitions have labels of the form $\langle a, n \rangle \star w$, where n is the time required to complete the standard action a at location w . We can easily modify α_l^p to obtain the abstraction function α_t^p that renders the considered timed transition systems. Assume as given a function $\text{time}(a)$ that assigns a duration to actions (we only consider visible actions, because the calculus of [19] has no τ 's, and we slightly change their notion of location). This abstraction function is as follows.

$$\alpha_t^p(X) = \{ \alpha_t^p(P \xrightarrow{\vartheta a} P') = \alpha_L(P) \xrightarrow{\langle a, n \rangle \star w} \alpha_{L'}(P') \mid P \xrightarrow{\vartheta a} P' \in X, \forall p : l \notin L_0^p \text{ and } \exists i : L_0^i = u, L_1^i = lu, \text{ and } \forall j \neq k : L_1^j = L_0^j, n = \text{time}(a) \}.$$

We have not yet considered transition systems expressing other qualitative aspects of concurrent processes. However, we are very confident that most of them

can be cast in our framework, and that abstract interpretation is a flexible tool for constructing specific *SOS* semantics for concurrency calculi, including those for mobility, like the π -calculus [27].

Acknowledgment. The authors wish to thank the referees for their precise remarks and helpful comments.

References

1. L. Aceto. A static view of localities. *Formal Aspects of Computing*, 1992.
2. R.M. Amadio and S. Prasad. Localities and failures. In *Proceedings of FST-TCS'94*, pages 205–216. Springer-Verlag, 1994.
3. M. Bernardo, L. Donatiello, and R. Gorrieri. Integrating performance and functional analysis of concurrent systems with EMPA. Technical Report UBLCS-95-14, University of Bologna, Laboratory for Computer Science, 1995.
4. C. Bodei, P. Degano, and C. Priami. Mobile processes with a distributed environment. In *Proceedings of ICALP'96, LNCS 1099*, pages 490–501. Springer-Verlag, 1996. To appear in TCS.
5. C. Bodei and C. Priami. True concurrency via abstract interpretation. In *Proceedings of SAS'97, LNCS 1302*, pages 202–216, 1997.
6. M. Boreale and D. Sangiorgi. A fully abstract semantics of causality in the π -calculus. In *Proceedings of STACS'95, LNCS*. Springer Verlag, 1995.
7. G. Boudol and I. Castellani. A non-interleaving semantics for CCS based on proved transitions. *Fundamenta Informaticae*, XI(4):433–452, 1988.
8. G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. A theory of processes with localities. *Theoretical Computer Science*, 114, 1993.
9. P. Buchholz. On a markovian process algebra. Technical report, Informatik IV, University of Dortmund, 1994.
10. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
11. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proceedings of POPL'92*, pages 83–94, 1992.
12. P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Procs. ICCL'94, IEEE*, pages 95–112, 1994.
13. P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint equational, constraint, closure-condition, rule-based and game-theoretic form. In *Proceedings of CAV'95, LNCS 939*, pages 293–308, 1995.
14. Ph. Darondeau and P. Degano. Causal trees. In *Proceedings of ICALP'89, LNCS 372*, pages 234–248. Springer-Verlag, 1989.
15. P. Degano, R. De Nicola, and U. Montanari. Partial ordering derivations for CCS. In *Proceedings of FCT, LNCS 199*, pages 520–533. Springer-Verlag, 1985.
16. P. Degano, J.-V. Loddo, and C. Priami. Mobile processes with local clocks. In *Proceedings of Workshop on Analysis and Verification of Multiple-Agent Languages*, Stockholm, Sweden, 1996.
17. P. Degano and C. Priami. Proved trees. In *Proceedings of ICALP'92, LNCS 623*, pages 629–640. Springer-Verlag, 1992.
18. P. Degano and C. Priami. Non interleaving semantics for mobile processes. Extended abstract. In *Proceedings of ICALP'95, LNCS 944*, pages 660–671. Springer-Verlag, 1995. To appear in TCS.

19. R. Gorrieri, M. Roccetti, and E. Stancapiano. A theory of processes with durational actions. *Theoretical Computer Science*, (140), 1994.
20. N. Götz, U. Herzog, and M. Rettelsbach. TIPP- a language for timed processes and performance evaluation. Technical Report 4/92, IMMD VII, University of Erlangen-Nurnberg, 1992.
21. E. Goubault. Durations for truly concurrent transitions. In *Proceedings of ESOP96, LNCS 1058*, pages 173–187, 1995.
22. M. Hennessy and T. Regan. A temporal process algebra. Technical Report 2/90, University of Sussex, 1990.
23. J. Hillston. The nature of synchronization. In U. Herzog and M. Rettelsbach, editors, *Proceedings of PAPM'94*, University of Erlangen, 1994.
24. A. Kiehn. Comparing causality and locality based equivalences. *Acta Informatica*, 31(8):697–718, 1994.
25. K.G. Larsen and A. Skou. Compositional verification of probabilistic processes. In *Proceedings of CONCUR'92*, volume 630 of *LNCS*. Springer-Verlag, 1992.
26. R. Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.
27. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (I and II). *Information and Computation*, 100(1):1–77, 1992.
28. U. Montanari and M. Pistore. Concurrent semantics for the π -calculus. In *Electronic Notes in Computer Science*, number 1. Elsevier, 1995.
29. U. Montanari and D. Yankelevich. A parametric approach to localities. In *Proceedings of ICALP'92, LNCS 623*, pages 617–628. Springer-Verlag, 1992.
30. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Real Time: Theory in Practice, LNCS 600*, pages 526–548. Springer-Verlag, 1991.
31. C. Priami. Stochastic π -calculus. *The Computer Journal*, 38(6):578–589, 1995.
32. C. Priami. *Enhanced Operational Semantics for Concurrency*. PhD thesis, Dipartimento di Informatica, Università di Pisa, March 1996. Available as Tech. Rep. TD-08/96.
33. C. Priami. Interleaving-based partial ordering semantics. In *Proceedings of Italian Conference on Theoretical Computer Science*, pages 264–278, Ravello, November 1995, 1996. World Scientific.
34. R.J. van Glabbeek, S.A. Smolka, B. Steffen, and C.M.N. Tofts. Reactive, generative and stratified models of probabilistic processes. *Information and Computation*, 1995.

A First-Order Language for Expressing Aliasing and Type Properties of Logic Programs

Paolo Volpe

Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
e-mail: volpep@di.unipi.it
tel: +39-50-887248. fax: +39-50-887226.

Abstract. In this paper we study a first-order language that allows to express and prove properties regarding the sharing of variables between non-ground terms and their types. The class of true formulas is proven to be decidable through a procedure of elimination of quantifiers and the language, with its proof procedure, is shown to have interesting applications in validation and debugging of logic programs. An interesting parallel is pointed out between the language of aliasing properties and the first order theories of Boolean algebras.

Keywords: *Verification of logic programs, languages of specification, first-order logic.*

1 Introduction

In many approaches to the verification of properties of logic programs, a formal language is required that allows to express the properties of programs one is interested in. In the methods proposed in [13][1][12][16], an assertional language is assumed to verify properties of arguments of predicates of the programs. Some verification conditions are provided that imply the partial correctness of the programs with respect to various aspects of the computations. For example the method proposed in [12] and [6] allows to prove properties of the correct answers of the programs, while in [16] a method is provided to prove properties of the computed answers. The methods proposed in [13] and [1] allow to prove, in addition, that predicates verify given specifications at call time.

In this paper we study a language that allows to express an interesting class of properties of non-ground terms, that is the data on which logic program operate. The language is sufficiently expressive to capture sharing, freeness and types of non-ground terms. Two or more terms are said to *share*, when they have at least one variable in common, while a term is *free* when it is a simple variable. For *types*, we will refer to term properties like being a list, a tree, a list of ground terms, etc.

Fragments of this language have already been studied in [18] and [2] and shown to be decidable, but in this paper we show that the full first-order theory is decidable. This allows to use its full expressive power in existing proof methods

and algorithmically decide whether the verification conditions are true or not. Indeed for many methods the verification conditions are expressed by formulas of the language. If the verification condition is true, we show the partial correctness of logic programs with respect to a property belonging to the class of aliasing properties and type assertions. If the verification condition is false (and we stress that this can be checked in finite time), obviously this does not mean that the program has necessarily an error. Anyway a “warning” can be raised up, signalling a possible wrong situation. The proof procedure shown in the paper can be easily enriched so as to provide a counterexample in this case. This allows the user to have more information about the warning and to decide whether to change the program (the counterexample is actually a wrong computation of the program) or to refine the specification (the verification condition is false because the specification is too “loose” and impossible computations are considered).

We want to emphasize the independent importance of the proof of decidability of the language. It is based on the method of elimination of quantifiers and points out an interesting set of formulas, which can be viewed as expressing constraints on the cardinality of the sets of variables that can occur in terms. Our proof is based on the parallel between the satisfiability of formulas of our language and the satisfiability of such cardinality constraints, which can be proven decidable as a consequence of the decidability of the theory of Boolean algebras. We think that such class of constraints, which has a quite simple representation and operations of composition and cylindrification, can be of interest in program analysis. For example, well known abstract domains such as *POS* [9] and *Sharing*[14] can be naturally viewed as subdomain of the class of cardinality constraints (see also [19]), with their composition and the cylindrification operator obtained as instances of the general ones.

The paper is organized as follows. In Section (2) we lay down the basic terminology. In Section (3) we define the class of types from which type assertions are built. In Section (4) we define formally the language, parametrically with respect to a family of regular types, and then we prove in Section (5) that such a language is decidable. Finally in Section (6) we give examples of application in the context of inductive proof methods.

2 Preliminaries

Throughout the paper, we assume familiarity with standard notions of logic programming and mathematical logic [17, 15].

A *first order language* $\mathcal{L} = \langle \Sigma, \Pi, \mathcal{V} \rangle$ is based on a set Σ of *function symbols*, a set Π of *predicate symbols* of assigned *arities* and a set \mathcal{V} of *variables*. The set $Terms(\Sigma, \mathcal{V})$ of all *terms* of \mathcal{L} with variables in \mathcal{V} is defined as usual. The set $Terms(\Sigma, \emptyset) \subseteq Terms(\Sigma, \mathcal{V})$ is the subset of *ground terms* (i.e. not containing variables). We write f, g for function symbols, p, q for predicate symbols, X, Y for variables, \mathbf{X} for tuples of distinct variables, t, s for terms, \mathbf{t}, \mathbf{s} for tuples of terms. A (predicate or function) symbol f with arity n will be denoted by $f^{(n)}$. *Atomic formulas* (also called *atoms*) are formulas like $p(t_1, \dots, t_n)$, with

$p^{(n)} \in \Pi$ and $t_1, \dots, t_n \in \text{Terms}(\Sigma, \mathcal{V})$. The set of *formulas* of \mathcal{L} is the smallest set \mathcal{F} containing the atomic formulas and such that if ϕ and ψ are in \mathcal{F} then $\neg\phi$, $\phi \wedge \psi$ and $\exists X\phi$, with $X \in \mathcal{V}$, are in \mathcal{F} . We will use the notation $\phi \vee \psi$ as a shorthand for $\neg(\neg\phi \wedge \neg\psi)$, $\phi \Rightarrow \psi$ for $\neg\phi \vee \psi$, $\phi \Leftrightarrow \psi$ for $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$ and $\forall X\phi$ for $\neg(\exists X\neg\phi)$. We assume also the constants *true* and *false* to be in \mathcal{F} . Given a syntactic object \mathcal{O} of \mathcal{L} , $\text{Vars}(\mathcal{O})$ denotes the set of free variables (not bound by any quantifier) of \mathcal{V} in \mathcal{O} . *Substitutions* are defined as mapping $\theta : \mathcal{V} \rightarrow \text{Terms}(\Sigma, \mathcal{V})$, which differ from the identity just on a finite subset of \mathcal{V} . They can be extended homomorphically to functions on terms. A preorder can be introduced on $\text{Terms}(\Sigma, \mathcal{V})$. Given $t, s \in \text{Terms}(\Sigma, \mathcal{V})$, $t \leq s$ iff there exists substitution η such that $t\eta = s$. The induced equivalence on terms is called *variance*.

An *interpretation* $\mathcal{I} = \langle \mathbf{D}, \Lambda, \Gamma \rangle$ of \mathcal{L} consists of a non-empty set \mathbf{D} , the *domain*; a set of functions $\Lambda_f : \mathbf{D}^n \rightarrow \mathbf{D}$ for each function symbol $f^{(n)} \in \Sigma$; a family of subsets $\Gamma_p \subseteq \mathbf{D}^n$ for each predicate symbol $p^{(n)} \in \Pi$. A *variable assignment* (also called *state*) $\sigma : \mathcal{V} \rightarrow \mathbf{D}$ maps each variable into an element of \mathbf{D} . It can be lifted homomorphically to a function, still denoted by σ , which maps terms in $\text{Terms}(\Sigma, \mathcal{V})$ to elements of \mathbf{D} . An atom $p(t_1, \dots, t_n)$ is *true in* \mathcal{I} *under the state* σ , written $\mathcal{I} \models_\sigma p(t_1, \dots, t_n)$, iff $(\sigma(t_1), \dots, \sigma(t_n)) \in \Gamma_p$. The *truth* of each formula of \mathcal{L} *under the state* σ , written $\mathcal{I} \models_\sigma \varphi$, is defined, as usual, by induction on the structure of φ . A formula φ is *true in* \mathcal{I} , that is $\mathcal{I} \models \varphi$, if and only if for each state $\sigma : \mathcal{V} \rightarrow \mathbf{D}$, $\mathcal{I} \models_\sigma \varphi$. The set $\text{Th}_{\mathcal{L}}(\mathcal{I})$ is defined as the set of all formulas of \mathcal{L} true in \mathcal{I} , that is as the set $\{\varphi \mid \varphi \text{ formula of } \mathcal{L} \text{ and } \mathcal{I} \models \varphi\}$.

In this paper we will be mainly interested in the non ground *term interpretations* $\mathcal{H} = \langle \text{Terms}(\Sigma, \mathcal{V}), \Lambda, \Gamma \rangle$, with the set $\text{Terms}(\Sigma, \mathcal{V})$ as domain and with $\Lambda_f(t_1, \dots, t_n) = f(t_1, \dots, t_n)$ for each $f^{(n)} \in \Sigma$ and tuple t_1, \dots, t_n of $\text{Terms}(\Sigma, \mathcal{V})$. In practice every (ground) term is interpreted by itself. We have chosen not to distinguish between the variables in formulas and the variables in the model. Indeed their roles are quite different and in practice no ambiguity arises. The preorder on $\text{Terms}(\Sigma, \mathcal{V})$ induces a preorder on the states $\sigma : \mathcal{V} \rightarrow \text{Terms}(\Sigma, \mathcal{V})$. The induced equivalence between states is still called *variance*.

3 Regular term grammars

To specify families of types we will consider regular term grammars. There is a large amount of papers on regular types. They have proved them to be a good trade-off between expressibility and decidability. In fact they are strictly more expressive than regular languages, but strictly contained in context-free languages (which have an undecidable subset relation). Our main references are the papers of Dart and Zobel [10, 11] and Boye and Maluszyński [3, 2].

A *regular term grammar* is a tuple $G = (\Sigma, \mathcal{V}, \mathbb{T}, \mathbb{R})$, where Σ is a set of function symbols, \mathcal{V} is an infinite denumerable set of variables, \mathbb{T} is a finite set of *type symbols*, including *var* and *any*, and \mathbb{R} is a finite set of rules $l \rightarrow r$ where

$$- l \in (\mathbb{T} \setminus \{\text{var}, \text{any}\})$$

– $r \in \text{Terms}(\Sigma, \mathbb{T})$

For every $T \in \mathbb{T} \setminus \{\text{var}, \text{any}\}$, we define $\text{Def}_G(T)$ (also denoted by $\text{Def}_{\mathbb{R}}(T)$) as the set $\{r \mid T \rightarrow r \in \mathbb{R}\}$. $\text{Def}_G(\text{var})$ is defined as the set of variable \mathcal{V} , while $\text{Def}_G(\text{any})$ as the set $\text{Terms}(\Sigma, \mathcal{V})$. We use the notation $T_1 \rightarrow_G T_2$ if T_2 is obtained from T_1 by replacing a symbol $T \in \mathbb{T}$ by a term in $\text{Def}_G(T)$. Let \rightarrow_G be the transitive and reflexive closure of \rightarrow_G . Given the type symbol $T \in \mathbb{T}$, we define the set of terms $[T]_G$, the *type* T , as the set $\{s \in \text{Terms}(\Sigma, \mathcal{V}) \mid T \rightarrow_G s\}$. Notice that $[\text{var}]_G = \mathcal{V}$ and $[\text{any}]_G = \text{Terms}(\Sigma, \mathcal{V})$. We assume function symbols in Σ to contain at least a constant and a function of arity 2. We will often omit the subscript when the grammar is clear from the context.

Example 1. Let us see some examples (taken in part from [2]) of regular types and of the grammars that define them. Let us suppose that $\Sigma = \{f_1^{(k_1)}, \dots, f_n^{(k_n)}\}$, where each function symbol $f_i^{(k_i)}$ has arity k_i . The set of ground and instantiated terms can be defined as:

$$\begin{array}{ll} \text{ground} \rightarrow f_1^{(0)} & \text{inst} \rightarrow f_1^{(0)} \\ \vdots & \vdots \\ \text{ground} \rightarrow f_{i-1}^{(0)} & \text{inst} \rightarrow f_{i-1}^{(0)} \\ \text{ground} \rightarrow f_i^{(k_i)}(\text{ground}, \dots, \text{ground}) & \text{inst} \rightarrow f_i^{(k_i)}(\text{any}, \dots, \text{any}) \\ \vdots & \vdots \\ \text{ground} \rightarrow f_n^{(k_n)}(\text{ground}, \dots, \text{ground}) & \text{inst} \rightarrow f_n^{(k_n)}(\text{any}, \dots, \text{any}) \end{array}$$

The type of lists, the lists of instantiated terms, the ground lists and the list of variables can be defined as :

$$\begin{array}{llll} \text{list} \rightarrow [] & \text{ilist} \rightarrow [] & \text{glist} \rightarrow [] & \text{vlist} \rightarrow [] \\ \text{list} \rightarrow [\text{any} \mid \text{list}] & \text{ilist} \rightarrow [\text{inst} \mid \text{ilist}] & \text{glist} \rightarrow [\text{ground} \mid \text{glist}] & \text{vlist} \rightarrow [\text{var} \mid \text{vlist}] \end{array}$$

Notice that if the type symbol var is available, the type any can be defined by

$$\begin{array}{ll} \text{any} \rightarrow \text{var} & \\ \text{any} \rightarrow f_1^{(0)} & \text{any} \rightarrow f_i^{(k_i)}(\text{any}, \dots, \text{any}) \\ \vdots & \vdots \\ \text{any} \rightarrow f_{i-1}^{(0)} & \text{any} \rightarrow f_n^{(k_n)}(\text{any}, \dots, \text{any}) \end{array}$$

The only reason to retain it, is that in closed grammars the type var is no longer available, but we still want to define non-ground types.

Regular term grammars enjoy several remarkable properties. The following lemmas can be shown, by slightly generalizing results and algorithms given in [10] and in [2].

Theorem 1 ([10]). *Given a regular term grammar G and a symbol type T , the set $[T]_G$ is decidable.*

Lemma 1 ([10]). *The emptiness and the subset relation of regular types is decidable.*

Indeed, given a grammar G and two type symbols T and S , it is possible to extend G into G' , with a new symbol $S \cap T$ and new rules in such a way that $[S \cap T]_{G'} = [S]_G \cap [T]_G$.

Lemma 2 ([10]). *There exists an algorithm that computes the intersection of regular types*

In this paper we will be mainly concerned with *closed discriminative* regular grammars in *normal form*.

Definition 1. *A regular term grammar $G = (\Sigma, \mathcal{V}, \mathbb{T}, \mathbb{R})$ is in normal form if each rule have the form $T \rightarrow \text{var}$ or*

$$T \rightarrow f(T_1, \dots, T_n)$$

with $f^{(n)} \in \Sigma$, $T \in \mathbb{T} \setminus \{\text{var}, \text{any}\}$ and $T_1, \dots, T_n \in \mathbb{T}$.

It can be easily shown that each type can be defined by a grammar in normal form.

Definition 2. *A regular term grammar G is discriminative if it is in normal form and, for each type symbol T , the top functors in $\text{Def}_G(T)$ are pairwise distinct.*

Definition 3. *A regular term grammar is closed if it is in normal form and, for each type symbol T , the symbol var does not occur in any element of $\text{Def}_G(T)$.*

Notice that most of the types used in logic programming allow closed and discriminative term grammars. For example, all the grammars introduced in (1), but for *vlist*, are discriminative and closed. It can be easily shown that regular types defined by a closed grammar are indeed closed under substitution.

Theorem 2 ([2]). *Given a closed regular term grammar G and a symbol type T , the set $[T]_G$ is closed under substitution.*

The types that can be defined by a discriminative and closed grammar will be referred to as *simple types*. Notice that if S and T are simple types, the intersection type $S \cap T$ is still simple.

An operation on types which we will need in the following is the difference of types. Given a discriminative and closed regular term grammar G and two type symbols T and S , we want to extend it to G' with a new symbol T/S and new rules in such a way that $[T \setminus S]_{G'} = [T]_G \setminus [S]_G$, if it is not the case that $[T]_G \subseteq [S]_G$. In general the grammar G' need not to be nor closed nor discriminative in general.

Example 2. Consider the difference type *any/inst*. It can easily be checked that the set $[\text{any}] / [\text{inst}]$ is equal to the set of variables \mathcal{V} . Anyway there is no closed grammar for such set of terms.

We will provide an algorithm which computes the difference T/S , assuming that type S is simple. The algorithm for the general case can be defined but it is more complex and we do not need such a generality. In fact we just need to compute differences types like $T_1 \cap \dots \cap T_n \setminus S_1 \setminus \dots \setminus S_k$, where T_1, \dots, T_n and S_1, \dots, S_k are simple types.

Difference Algorithm

INPUT. Two type symbols T and S and the set \mathbb{R} of rules defining T and S , with S simple.

OUTPUT. A pair $(T \setminus S, \mathbb{S})$, where $T \setminus S$ is defined by the rules in \mathbb{S} , if $\mathbb{S} \neq \emptyset$; otherwise the difference is empty ($[T] \subseteq [S]$).

METHOD. The algorithm is defined by the following recursive function. A set I of difference symbols $T \setminus S$ is used to ensure termination. The type symbol *any* is supposed to be unfolded as in example (1).

$$\begin{aligned} \text{difference}(T, S, \mathbb{R}) &= \text{difference}(T, S, \mathbb{R}, \emptyset) \\ \text{difference}(T, S, \mathbb{R}, I) &= \end{aligned}$$

- If $T \subseteq S$ then return $(T \setminus S, \emptyset)$;
 - If the symbol $T \setminus S$ is in I then return $(T \setminus S, \mathbb{R})$;
 - Otherwise, let $\text{Def}_{\mathbb{R}}(T) = \{r_1, \dots, r_k\}$. For each $i \in \{1, \dots, k\}$, let H_i be defined as follows:
 - if $r_i = \text{var}$ or $(r_i = f_i(T_1, \dots, T_{n_i})$ and the functor f_i does not occur in $\text{Def}_{\mathbb{R}}(S)$) then let $H_i = \{(T \setminus S) \rightarrow r_i\}$;
 - If $r_i = f_i(T_1, \dots, T_{n_i})$ and $f_i(S_1, \dots, S_{n_i}) \in \text{Def}_{\mathbb{R}}(S)$ then let $(T_j \setminus S_j, \mathbb{S}_j) = \text{difference}(T_j, S_j, \mathbb{R}, I \cup \{T \setminus S\})$, for each $j = \{1, \dots, n_i\}$, and let $H_i = \bigcup_{\mathbb{S}_j \neq \emptyset} \{(T \setminus S) \rightarrow f_i(T_1, \dots, T_j \setminus S_j, \dots, T_{n_i})\} \cup \mathbb{S}_j$
- Return $(T \setminus S, \mathbb{R} \cup \bigcup_{i=1}^k H_i)$

Lemma 3. *Let T and S be two type terms defined by the rules of \mathbb{R} , S simple. Then $\text{difference}(T, S, \mathbb{R})$ terminates and returns a pair $(T \setminus S, \mathbb{S})$ such that $[T \setminus S]_{\mathbb{S}} = [T]_{\mathbb{R}} \setminus [S]_{\mathbb{R}}$.*

To carry on the elimination of quantifiers in the next section, we need to know the cardinalities of the sets of variables which may occur in a term of a given type.

Definition 4. *Given a type symbol T defined by a grammar G , the var-cardinality of T , written $|T|$, is defined as the set $\{|\text{Vars}(t)| \mid t \in [T]_G\}$.*

In other terms, $k \in |T|$ if and only if there exists $t \in [T]$ such that $|\text{Vars}(t)| = k$. We can prove in a straightforward way the following lemmas.

Lemma 4. *Given a simple type T , then $|T|$ is equal to $\{0\}$ or to ω . It is decidable which is the case.*

Proof. The relation $T \subseteq \text{ground}$, with ground as defined in example (1), is decidable by lemma (1). If it is the case then $|T| = \{0\}$, otherwise $|T| = \omega$, since T is substitution closed, the signature Σ contains at least a function symbol of arity 2 and a constant, and the set of variables is infinite denumerable.

For difference types things can be more complex.

Example 3. Consider again the difference type $\text{any} \backslash \text{inst}$. It can be easily seen that $|\text{any} \backslash \text{inst}|$ is the set $\{1\}$.

Anyway the behaviour of difference types $T_1 \cap \dots \cap T_n \backslash S_1 \setminus \dots \setminus S_k$, with T_1, \dots, T_n and S_1, \dots, S_k simple types, is sufficiently regular so as to show the following theorem.

Theorem 3. *The var-cardinality of the type $T_1 \cap \dots \cap T_n \backslash S_1 \setminus \dots \setminus S_k$, with T_1, \dots, T_n and S_1, \dots, S_k simple types, is equal to $S \cup [k, \omega]$, where S is a finite set of natural number and $[k, \omega]$, with $k = 1, \dots, \omega$ is the set of natural numbers greater or equal to k .*

The proof provides an effective procedure to compute the var-cardinality of a difference type.

4 A language of properties

In this section we introduce a language that allows to express properties of terms used in static analysis and verification of logic programs: these include groundness, freeness, sharing, type assertions. The language is parametric with respect to a family of types defined through a regular term grammar. It is an extension of the language proposed by Marchiori in [18].

We assume a regular term grammar $G = (\Sigma, \mathcal{V}, \mathbb{T}, \mathbb{R})$, discriminative and closed, describing the family of types we are interested in. As before, the set of function symbols Σ is assumed to contain at least a constant and a function of arity 2, \mathcal{V} is assumed to be a denumerable set of variables. We define then a first-order language $\mathcal{L}_G = \langle \Sigma, \Pi, \mathcal{V} \rangle$, starting from the regular term grammar G . The set of predicate symbol Π consists of the predicates $\text{var}^{(1)}$, $\text{share}^{(n)}$, for each natural n , and a unary predicate $p_T^{(1)}$ for each symbol type $T \in \mathbb{T} \setminus \{\text{any}\}$. Often we will write $T(t)$ for the atomic formula $p_T(t)$. For example, $p_{\text{ground}}(t)$ will be often written as $\text{ground}(t)$. We will omit the subscript G in \mathcal{L}_G , when no confusion arises.

Like in [18], we give the semantics of formulas \mathcal{L} by considering the non-ground Herbrand interpretation $\mathcal{H} = \langle \text{Terms}(\Sigma, \mathcal{V}), \Lambda, \Gamma \rangle$. We define the interpretation Γ for predicate symbols directly through the truth relation \models_σ . Given a state σ , the relation \models_σ is defined on atoms as follows.

- $\mathcal{H} \models_\sigma \text{var}(t)$ iff $\sigma(t) \in \mathcal{V}$;
- $\mathcal{H} \models_\sigma \text{share}(t_1, \dots, t_n)$ iff $\bigcap_{i=1}^n \text{Vars}(\sigma(t_i)) \neq \emptyset$;
- $\mathcal{H} \models_\sigma p_T(t)$ iff $\sigma(t) \in [T]_G$, with $T \in \mathbb{T} \setminus \{\text{any}\}$.

The semantics of the other formulas of \mathcal{L} can be derived as usual. We will often write $\models \varphi$ (resp. $\models_\sigma \varphi$) for $\mathcal{H} \models \varphi$ (resp. $\mathcal{H} \models_\sigma \varphi$).

Example 4. Let us see examples of the expressive power of \mathcal{L}_G .

- The formula $\forall V \text{ var}(V) \Rightarrow \neg \text{share}(V, X)$ asserts the groundness of X ;
- the formula $\text{list}(X) \wedge \exists V \text{ var}(V) \wedge \text{share}(V, X) \wedge (\forall W \text{ var}(W) \wedge \text{share}(W, X) \Rightarrow \text{share}(V, W))$ says that X is a list in which exactly one variable occurs;
- $\forall V \text{ var}(V) \wedge \text{share}(V, Y) \Rightarrow \text{share}(V, X)$ says that each variable in Y is also in X ;
- $(\forall V \text{ var}(V) \wedge \text{share}(V, Y) \Rightarrow \text{share}(V, X)) \wedge \text{ground}(X) \Rightarrow \text{ground}(Y)$ asserts that if $\forall V \text{ var}(V) \wedge \text{share}(V, Y) \Rightarrow \text{share}(V, X)$ (i.e. $\text{Vars}(Y) \subseteq \text{Vars}(X)$) and $\text{ground}(X)$ (i.e. $\text{Vars}(X) = \emptyset$) then $\text{ground}(Y)$ (i.e. $\text{Vars}(Y) = \emptyset$).

Notice that properties expressible in the language \mathcal{L}_G are invariant with respect to the name of variables. That is, intuitively if property φ is true about $t \in \text{Terms}(\Sigma, \mathcal{V})$ and s is a variant of t then φ is true of s , too. More formally the following lemma can be shown.

Lemma 5. *For each φ , formula of \mathcal{L} , if $\models_\sigma \varphi$ and σ' is a variant of σ then $\models_{\sigma'} \varphi$.*

An important class of formulas of \mathcal{L}_G , which are often considered in analysis and verification, is the class of *monotone formulas*, that is the formulas φ such that $\models_\sigma \varphi$ and $\sigma \leq \sigma'$ implies $\models_{\sigma'} \varphi$. For example, $\text{ground}(X)$ and $\neg \text{var}(X)$ are monotone properties, while $\text{var}(X)$ and $\text{share}(X, Y)$ are not. Since the grammar G is closed, by lemma (2), each atom $p_{T_1}(X)$ is monotone. An interesting subclass of monotone properties are the dependences like $\forall V \text{ var}(V) \wedge \text{share}(V, Y) \Rightarrow \text{share}(V, X)$. This kind of formulas are used in logic programming analysis since they allow to relate the values to which two or more arguments of a predicate can be instantiated. As shown by the example, the formula $\forall V \text{ var}(V) \wedge \text{share}(V, Y) \Rightarrow \text{share}(V, X)$, could be read informally as saying that if X is instantiated to a ground value then also Y is.

The class of monotone properties is closed with respect to the connectives \wedge, \vee and the quantifiers. We think it would be very interesting to give a complete syntactical characterization of these properties inside \mathcal{L} .

5 A proof procedure for \mathcal{L}

We are interested in characterizing the set of formulas $\text{Th}_{\mathcal{L}}(\mathcal{H})$, that is the formulas of the language \mathcal{L} which are true in the interpretation \mathcal{H} .

It is known that the existential fragment of the language \mathcal{L} without the type predicates is decidable. In fact in [18], it is proposed a proof procedure to decide the validity of formulas $\exists(\varphi_1 \wedge \dots \wedge \varphi_n)$ where each atom φ_i is an atom $\text{var}(t)$, $\text{ground}(t)$, $\text{share}(t_1, \dots, t_n)$, or its negation. It is also known that the implication between regular types is decidable. In fact in [3, 2] a procedure is proposed to decide the validity of implications like $\forall(p_{T_1}(t_1) \wedge \dots \wedge p_{T_n}(t_n) \Rightarrow p_T(t))$. It is not

clear whether putting them together and considering the full first order theory, such as in \mathcal{L} , the language is still decidable. In this section we will show that this is indeed the case and it is not such a trivial extension of those previous results.

To show that $Th_{\mathcal{L}}(\mathcal{H})$ is recursive we will use the method of *elimination of quantifiers* [5, 15]. We single out a set Ω , the *elimination set*, of fomulas of \mathcal{L} and show that each formula of \mathcal{L} is equivalent in \mathcal{H} to a boolean combination of formulas of Ω . Once proven the decidability of formulas in Ω , we end up with a complete decision procedure for formulas of \mathcal{L} .

In the following, we use the abbreviation $\exists_{\geq k} var(V) \phi$ to say that there exist at least k distinct variables which verify formula ϕ . It is defined by induction on k .

$$\begin{aligned} \exists_{\geq 1} var(V) \phi &\text{ is } \exists V \, var(V) \wedge \phi \\ \exists_{\geq k+1} var(V) \phi &\text{ is } \exists V \, var(V) \wedge \phi \wedge (\exists_{\geq k} var(W) \phi [V \setminus W] \wedge \neg share(V, W)), \end{aligned}$$

where the formula $\phi [V \setminus W]$ is obtained by ϕ by replacing all occurrence of V with W .

To carry on the elimination of quantifiers we will need a particular class of formulas, which we call cardinality constraints.

Definition 5. Let \mathcal{B} be the class of boolean terms on \mathcal{V} , that is the terms built from the signature $(\{\cap^{(2)}, \cup^{(2)}, \neg^{(1)}, 0^{(0)}, 1^{(0)}\}, \mathcal{V})$. Fixed a natural k and a boolean term t , a simple cardinality constraint $\alpha_k(t)$ is defined as the formula $\exists_{\geq k} var(V) \Psi_t(V)$, where $\Psi_t(V)$ is defined inductively on the syntax of t .

- $\Psi_0(V) = \text{false}$ and $\Psi_1(V) = \text{true}$;
- $\Psi_X(V) = share(V, X)$, with $X \in \mathcal{V}$;
- $\Psi_{t_1 \cap t_2}(V) = \Psi_{t_1}(V) \wedge \Psi_{t_2}(V)$;
- $\Psi_{t_1 \cup t_2}(V) = \Psi_{t_1}(V) \vee \Psi_{t_2}(V)$;
- $\Psi_{\neg t}(V) = \neg \Psi_t(V)$.

A simple cardinality constraint $\alpha_k(t)$ asserts the membership of at least k elements to the combination of variables in t , seen as subsets of \mathcal{V} . Often the term $\neg t$ will be written as \bar{t} .

Example 5. Consider the simple cardinality constraint $\alpha_3(X \cap Y \cap \bar{Z})$, that is, the formula $\exists_{\geq 3} var(V) share(V, X) \wedge share(V, Y) \wedge \neg share(V, Z)$. This formula is true in \mathcal{H} under the state σ , if there exist at least three variables sharing with $\sigma(X)$ and $\sigma(Y)$ and not with $\sigma(Z)$, that is if the cardinality of $Vars(\sigma(X)) \cap Vars(\sigma(Y)) \cap \overline{Vars(\sigma(Z))}$, is at least equal to 3.

The following lemma allows us to work with simple cardinality constraints just as if they were assertions on set of variables.

Lemma 6. Let $\sigma : \mathcal{V} \rightarrow Terms(\Sigma, \mathcal{V})$. Let t_{σ}^* be obtained by the boolean term t by replacing each occurrence of a variable X with $Vars(\sigma(X))$, for each $X \in \mathcal{V}$. Then $\models_{\sigma} \alpha_k(t)$ if and only if the cardinality of t_{σ}^* is at least equal to k .

Proof. It can be straightforwardly proved by induction on t and k , noting that, if $W \in \mathcal{V}$, then $\models_{\sigma[V \setminus W]} \Psi_t(V)$ if and only if $W \in t_{\sigma}^*$

Lemmas (7) and (8) follow.

Lemma 7. *Let $\sigma, \sigma' : \mathcal{V} \rightarrow \text{Terms}(\Sigma, \mathcal{V})$ two states such that, for each $X \in \mathcal{V}$, $\text{Vars}(\sigma(X)) = \text{Vars}(\sigma'(X))$. Then $\models_{\sigma} \alpha_k(t)$ if and only if $\models_{\sigma'} \alpha_k(t)$.*

Lemma 8. *Let t_1 and t_2 be two terms equivalent as boolean terms. Then $\models \alpha_k(t_1) \Leftrightarrow \alpha_k(t_2)$.*

We will use the formula $\alpha_{=k}(t)$, that is t contains exactly k elements, as an abbreviation for the formula $\alpha_k(t) \wedge \neg \alpha_{k+1}(t)$.

Let the elimination set Ω be composed by the atomic formulas $\text{var}(X)$, $p_{T_1}(X), \dots, p_{T_n}(X)$, where $X \in \mathcal{V}$, and by the set of simple cardinality constraints $\{\alpha_k(t) \mid k \geq 1, t \text{ is a boolean term}\}$. The idea is to exploit the striking similarity of the simple cardinality constraints in our language with formulas of the first-order theory of the powerset of \mathcal{V} seen as a Boolean algebra. For such a theory the decidability has been shown by Skolem in 1919 just through an argument based on elimination of quantifier (see [15] for a slightly more general account). The main idea is to reduce satisfiability of a formula in \mathcal{L} to satisfiability of a conjunction of cardinality constraints.

Definition 6. *A conjunction of simple cardinality constraints $\alpha_k(t)$, possibly negated, is a cardinality constraint.*

The elimination of quantifiers can be carried on for cardinality constraints in a long but straightforward way. The proof is adapted from the proof of elimination of quantifiers of the theory of Boolean algebras in [15].

Theorem 4. *Let $\psi(X)$ be a cardinality constraint. Then $\exists X \psi(X)$ is equivalent to a disjunction of cardinality constraints.*

Anyway, in general, in a formula of \mathcal{L} , other formulas than cardinality constraints may occur. We will show then some results that allow to eliminate formulas different from cardinality constraints under existential quantifiers.

Definition 7. *A formula is flat if it does not contain any functor.*

Notice that each formula $\alpha_k(t)$ is flat. Indeed we can consider only flat formulas as shown by next lemma.

Lemma 9. *Every formula φ of \mathcal{L} is equivalent in \mathcal{H} to a flat formula.*

Proof. The following equivalences in \mathcal{H} , already appeared in [18] and [2], can be easily checked.

- $\text{var}(f(X)) \Leftrightarrow \text{false}$ for each functor $f \in \Sigma$;
- $\text{share}(t_1, \dots, f(s_1, \dots, s_k), \dots, t_n) \Leftrightarrow \bigvee_{i=1}^k \text{share}(t_1, \dots, s_i, \dots, t_n)$;
- $p_T(f(s_1, \dots, s_n)) \Leftrightarrow p_{T_1}(s_1) \wedge \dots \wedge p_{T_n}(s_n)$ for each $f(T_1, \dots, T_k) \in \text{Def}_G(T)$ (remember that G is discriminative).
- $p_T(f(s_1, \dots, s_n)) \Leftrightarrow \text{false}$ if $f(T_1, \dots, T_k) \notin \text{Def}_G(T)$ for any T_1, \dots, T_k .

Definition 8. A type formula is a conjunction of atomic flat formulas $p_T(X)$, $X \in \mathcal{V}$, possibly negated.

Type formulas can be eliminated under existential quantifier and substituted by cardinality constraints.

Lemma 10. Let $\psi(X)$ be a cardinality constraint and $\phi(X) = p_{T_1}(X) \wedge \cdots \wedge p_{T_n}(X) \wedge \neg p_{S_1}(X) \wedge \cdots \wedge \neg p_{S_k}(X)$ a type formula. Let $S \cup [k, \omega]$ be the var-cardinality of $T_1 \cap \cdots \cap T_n \setminus S_1 \setminus \cdots \setminus S_k$. Then the following are valid equivalences in \mathcal{H} .

- $(\exists X \phi(X) \wedge \psi(X)) \Leftrightarrow \text{false}$ if S is empty and $k = \omega$;
- $(\exists X \phi(X) \wedge \psi(X)) \Leftrightarrow (\bigvee_{h \in S} \exists X \alpha_{=h}(X) \wedge \psi(X)) \vee (\exists X \alpha_k(X) \wedge \psi(X))$.

Proof. It is a consequence of lemmas (3) and (7) and of $[T_1] \cap \cdots \cap [T_n] \cap [S_1] \cap \cdots \cap [S_k] = [T_1 \cap \cdots \cap T_n \setminus S_1 \setminus \cdots \setminus S_k]$.

In the same way the formulas $\text{var}(X)$ and $\neg \text{var}(X)$ can be eliminated.

Lemma 11. Let $\psi(X)$ be a cardinality constraint. Then $(\exists X \neg \text{var}(X) \wedge \psi(X)) \Leftrightarrow (\exists X \psi(X))$ and $(\exists X \text{var}(X) \wedge \psi(X)) \Leftrightarrow (\exists X \alpha_{=1}(X) \wedge \psi(X))$.

We can prove then the following theorem, which is at the base of the procedure of elimination of quantifiers.

Theorem 5. For each formula $\Psi(X)$, conjunction of formulas of Ω , possibly negated, there exists Φ , a boolean composition of formulas of Ω , such that $\models (\exists X \Psi(X)) \Leftrightarrow \Phi$.

Proof. Let $\Psi(X)$ be a conjunction of formulas of Ω , possibly negated. We can write $\Psi(X) = \Psi_1(X) \wedge \Psi_2(X)$, with $\Psi_1(X)$ the conjunction of all simple cardinality constraints in $\Psi(X)$ and $\Psi_2(X)$ the conjunction of the remaining formulas. We can assume that $\text{var}(X)$ or $\neg \text{var}(X)$ appears in $\Psi_2(X)$.

In the first case we can assume that no other formula appears, that is $\Psi_2(X) = \text{var}(X)$. In fact if $p_T(X)$ appears in $\Psi_2(X)$, we have that $\Psi_2(X) \Leftrightarrow \text{false}$. Moreover we have the equivalence $\text{var}(X) \wedge \neg p_T(X) \Leftrightarrow \text{var}(X)$. We have then $\Psi(X) = \Psi_1(X) \wedge \text{var}(X)$, with $\Psi_1(X)$ a cardinality constraint. By lemma (11) we have that $\exists X \Psi_1(X) \wedge \text{var}(X) \Leftrightarrow \exists X \Psi_1(X) \wedge \alpha_{=1}(X)$.

In case $\neg \text{var}(X)$ appears in $\Psi_2(X)$ and no other formula occurs in $\Psi_2(X)$, we have $\Psi(X) = \Psi_1(X) \wedge \neg \text{var}(X)$ and $(\exists X \Psi_1(X) \wedge \neg \text{var}(X)) \Leftrightarrow \exists X \Psi_1(X)$. Otherwise we may assume that $\Psi_2(X)$ has the general form

$$\neg \text{var}(X) \wedge p_{T_1}(X) \wedge \cdots \wedge p_{T_n}(X) \wedge \neg p_{S_1}(X) \wedge \cdots \wedge \neg p_{S_k}(X),$$

where $n + k \geq 1$, which is equivalent to $p_{\text{inst}}(X) \wedge p_{T_1}(X) \wedge \cdots \wedge p_{T_n}(X) \wedge \neg p_{S_1}(X) \wedge \cdots \wedge \neg p_{S_k}(X)$. We have then $\Psi(X) = \Psi_1(X) \wedge (p_{\text{inst}}(X) \wedge p_{T_1}(X) \wedge \cdots \wedge p_{T_n}(X) \wedge \neg p_{S_1}(X) \wedge \cdots \wedge \neg p_{S_k}(X))$ with $\Psi_1(X)$ a cardinality constraint. By lemma (10) we know that $\exists X \Psi(X)$, is equivalent to false or to a disjunction of formulas $\exists X \Delta_i(X)$, where each $\Delta_i(X)$ is a cardinality constraint.

In any case, we have that $\exists X \Psi(X)$ is equivalent to a disjunction of formulas

$\exists X \Phi(X)$, with $\Phi(X)$ a cardinality constraint. We can apply then lemma (4) to each disjunct and obtain a boolean formula, which is a combination of formulas of Ω equivalent to $\exists X \Psi(X)$.

Notice that the procedure given in the proof of the previous theorem is effective. At this point it is easy to show that for each first-order formula of \mathcal{L} it can be computed a Boolean composition of formulas of Ω equivalent to it on \mathcal{H} .

Theorem 6. *Every formula Ψ of \mathcal{L} is equivalent in \mathcal{H} to a Boolean composition of formulas of Ω .*

Proof. Notice that since $share(X_1, \dots, X_n)$ and $\alpha_1(X_1 \cap \dots \cap X_n)$ are equivalent in \mathcal{H} , we can assume that no formula with predicate $share$ occurs in Ψ . The proof is by structural induction on Ψ . Every atom of \mathcal{L} is equivalent to a combination of atomic formulas of Ω (see proof of lemma (9)). If Ψ is a Boolean composition of formulas, then, by induction, each one of these is equivalent to a Boolean combination of formulas of Ω , and thus Ψ is equivalent to a Boolean combination of formulas of Ω . If Ψ is equal to $\exists X \Lambda$, then by induction Λ is equivalent to Λ^* , boolean combination of formulas of Ω . We put Λ^* in disjunctive normal form. The existential quantifier distributes over the disjunction and Ψ is equivalent to a disjunction of formulas $\exists X \Psi_i$ with each Ψ_i conjunction of atoms, possibly negated, of Ω . By the previous theorem, each one of these is equivalent to a boolean formula of Ω .

We have proven that each first-order formula of \mathcal{L} is equivalent to a boolean combination of formulas of Ω . With the next theorem we state explicitly the decidability of truth of formulas in Ω .

Theorem 7. *There exists an algorithm which decides the satisfiability of formulas $\bigwedge \varphi_i$, with each φ_i a formula of Ω , possibly negated.*

Proof. The formula is satisfiable in \mathcal{H} if $\exists \bigwedge \varphi_i$ is true on \mathcal{H} , where $\exists \bigwedge \varphi_i$ is the existential closure of $\bigwedge \varphi_i$. We can apply the elimination of quantifiers to $\exists \bigwedge \varphi_i$. Since the resulting formula has no variable, it must be equivalent to *true* or *false*.

Alternatively, an algorithm can be provided that, in case $\exists \bigwedge \varphi_i$ is satisfiable, find a substitution for the variables of $\bigwedge \varphi_i$. It can be obtained by first reducing $\exists \bigwedge \varphi_i$ to a disjunction of cardinality constraints by lemmas (10) and (11). An algorithm proposed in [4] to decide validity of a subclass of the language of set theory, can be instantiated to our case to solve cardinality constraints. Once found such solutions, we have, for each variable X of $\bigwedge \varphi_i$, the set of variables S_X of the term to which X must be mapped for $\bigwedge \varphi_i$ to be satisfied. At this point a term can be built for each such X , containing the variables in S_X and verifying the formulas in $\bigwedge \varphi_i$, that is verifying the assertion $var(X)$, $\neg var(X)$ or $p_{T_1}(X) \wedge \dots \wedge p_{T_n}(X) \wedge \neg p_{S_1}(X) \wedge \dots \wedge \neg p_{S_k}(X)$. In the first two cases the procedure is obvious. In the third case we use the rules that define regular type $T_1 \cap \dots \cap T_n \setminus S_1 \setminus \dots \setminus S_k$.

Now for Ψ a general combination of formulas of Ω , we have that $\forall\Psi \Leftrightarrow \forall\bigwedge\bigvee\varphi_i \Leftrightarrow \neg\exists\bigvee\bigwedge\neg\varphi_i \Leftrightarrow \neg\bigvee\exists(\bigwedge\neg\varphi_i)$, with each φ_i a formula of Ω , possibly negated. Then $\forall\Psi$ is true if and only if none of the conjunction $\bigwedge\neg\varphi_i$ is satisfiable. Notice that in case Ψ is not true that means that a conjunction $\bigwedge\neg\varphi_i$ is satisfiable and, by the previous theorem, a counterexample can be provided.

6 Applications

The logic we have studied is expressive enough to capture many interesting properties of arguments of predicates of logic programs. This suggests to employ \mathcal{L} as an assertional language to be used in inductive proof methods. We will show how the verification conditions of many such methods can be entirely expressed by a formula of \mathcal{L} . Moreover since \mathcal{L} is equipped with an algorithmic proof procedure, such verification conditions can effectively be decided.

Throughout the section we assume $\langle\Sigma, \Lambda, \mathcal{V}\rangle$ as the signature for logic programs, with Λ the set of predicate symbols, distinct from the predicates of \mathcal{L} . An assertion Θ of \mathcal{L} is said a *specification* for predicate $p^{(n)} \in \Lambda$, if $\text{Vars}(\Theta) \subseteq \{X_1, \dots, X_n\}$. Informally variable X_i refers to the i -th argument of p . An atom $p(t_1, \dots, t_n)$ will be said *to satisfy the assertion* Θ , written $p(t_1, \dots, t_n) \models \Theta$, iff $\mathcal{H} \models_{\sigma[X_1, \dots, X_n \setminus t_1, \dots, t_n]} \Theta$. The notation $\Theta[\mathbf{X} \setminus \mathbf{t}]$ denotes the formula Θ in which the variables (X_1, \dots, X_n) are simultaneously substituted by terms (t_1, \dots, t_n) .

Let us consider first the correct answers of a program. We associate a specification Θ_p to each predicate $p \in \Lambda$. Program P is *success-correct* with respect to $\{\Theta_p\}_{p \in \text{Pred}}$ iff $\forall p(t) \in \text{Atoms } p(t) \overset{\theta}{\rightsquigarrow} \square$ implies $p(t)\theta \models \Theta_p$. A sufficient condition for correctness can be stated as follows. A program P is success-correct with respect to $\{\Theta_p\}_{p \in \text{Pred}}$ if for each clause $p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \dots, p_n(\mathbf{t}_n)$ it is true that

$$\mathcal{H} \models \bigwedge_{i=1}^n \Theta_{p_i}[\mathbf{X}_i \setminus \mathbf{t}_i] \Rightarrow \Theta_p[\mathbf{X} \setminus \mathbf{t}]. \quad (1)$$

The method indeed has been proposed in [6] and [12]. In our case, condition (1) can be decided using the procedure of Section (5). If the formula (1) is proved to be true for each clause, then the program is partial correct. Obviously if the formula is false this does not imply that the clause is necessarily wrong. Anyway it could be considered a warning that something wrong can happen. To this aim the alternative algorithm proposed in the proof of Theorem (7), could be very useful, since it would allow to provide counterexamples in such cases. The user then would have more information to decide whether the warning can give raise to a real error or simply the specification is too loose and behaviours are considered that can never occur in practice.

If we are interested to check the Input/Output behaviour of logic programs, we can proceed as follows. To each predicate $p \in \Lambda$, is associated a property

$pre_p \rightarrow post_p$, where pre_p and $post_p$ are specifications for p . Now a program P is *I/O-correct* with respect to the properties $\{pre_p \rightarrow post_p\}_{p \in Pred}$ if

$$p(t) \models pre_p \text{ and } p(t) \overset{\theta}{\rightsquigarrow} \square \text{ implies } p(t)\theta \models post_p.$$

Now in case each formula pre_p is monotone, a sufficient condition for P to be *I/O-correct* with respect to the properties $\{pre_p \rightarrow post_p\}_{p \in Pred}$ is that

$$\mathcal{H} \models \left(\bigwedge_{i=1}^n (pre_{p_i} [\mathbf{X}_i \setminus \mathbf{t}_i] \Rightarrow post_{p_i} [\mathbf{X}_i \setminus \mathbf{t}_i]) \wedge pre_p [\mathbf{X} \setminus \mathbf{t}] \right) \Rightarrow post_p [\mathbf{X} \setminus \mathbf{t}].$$

This can be shown to correspond to a particular case of the previous method [16]. Anyway, we still have a formula of \mathcal{L} that can be decided algorithmically. Finally, if we want to check the call correctness of predicates we can consider methods like those proposed in [13][1]. Like in the previous case, to each predicate $p \in \Lambda$, is associated a property $pre_p \rightarrow post_p$, with pre_p and $post_p$ specifications for p . In this case, anyway, the pre-condition is used also as a specification for the argument of a predicate at call-time. In fact a program P is *call-correct* with respect to the properties $\{pre_p \rightarrow post_p\}_{p \in Pred}$ if

$$p(t) \models pre_p \text{ and } p(t) \overset{\theta}{\rightsquigarrow} \square \text{ implies } p(t)\theta \models post_p$$

and

$$p(t) \models pre_p \text{ and } p(t) \overset{*}{\rightarrow} \langle q(s), \mathbf{G} \rangle \text{ implies } q(s) \models pre_q.$$

We are assuming a leftmost selection rule for *SLD*-derivations. In [1] it has been shown that, in the case pre_p and $post_p$ are monotone for each p , then a sufficient condition for P to be call-correct with respect to $\{pre_p \rightarrow post_p\}_{p \in Pred}$ is that for each clause $p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \dots, p_n(\mathbf{t}_n)$, it is true that for each $1 \leq k \leq n+1$

$$\mathcal{H} \models (pre_p [\mathbf{X} \setminus \mathbf{t}] \wedge \bigwedge_{i=1}^{k-1} post_{p_i} [\mathbf{X}_i \setminus \mathbf{t}_i] \Rightarrow pre_{p_k} [\mathbf{X}_k \setminus \mathbf{t}_k])$$

where $pre_{p_{n+1}} [\mathbf{X}_{n+1} \setminus \mathbf{t}_{n+1}] \equiv post_p [\mathbf{X} \setminus \mathbf{t}]$.

In this case we have a finer control on the possible run-time behaviour. In fact a warning in this case can be raised because there may be a computation that calls a predicate with arguments which violate the specification. Again if a counterexample is provided the user may decide whether the specification is too loose or an actual error has been discovered.

We want to stress the restriction in previous methods to monotone properties. The reason for not considering all the expressible properties is that in those cases the verification conditions become much more complex and the mgu's have to be considered explicitly (see [16]). While for monotone assertions, the verification conditions are expressible as formulas of \mathcal{L} , this is no more the case if more general properties are considered. A solution we are working on, is to enrich \mathcal{L} so as to express formally the mgu's.

Anyway the class of properties that can be mechanically checked is still quite large, including type assertions, groundness, dependencies.

7 Conclusions

In this paper we have studied a language that allows to express and decide properties of finite terms on a given signature, that is the data on which logic programs operate. The language is able to express aliasing properties such as the sharing or freeness and is enriched with type assertions. Using formulas of \mathcal{L} , we are able to express many properties of program and prove them in inductive proof methods. Moreover since the logic is decidable we can mechanically check the corresponding verification condition and raise a warning if the condition is not verified. In such cases counterexamples can be built, which can be helpful for the user who is carrying on the verification.

The set of true formulas is proved to be decidable through a procedure of elimination of quantifiers. This points out an interesting class of formulas, which express cardinality constraints on the set of variables that can occur in a term. This can give an interesting insight on domains used for the analysis of logic program. In fact many of the domains used for aliasing analysis can be seen as fragments of the domain of formulas of cardinality constraints. For example the element $(X \wedge Y) \rightarrow (Z \vee W)$ of \mathcal{POS} [9], can be represented as $(Vars(Z) \subseteq Vars(X) \cup Vars(Y)) \vee (Vars(W) \subseteq Vars(X) \cup Vars(Y))$, which is equivalent to $(|Vars(Z) \cap \overline{Vars(X)} \cap \overline{Vars(Y)}| = 0) \vee (|Vars(W) \cap (\overline{Vars(X)} \cap \overline{Vars(Y)})| = 0)$, that is the constraint $\alpha_{=0}(Z \cap \overline{X} \cap \overline{Y}) \vee \alpha_{=0}(W \cap \overline{X} \cap \overline{Y})$ of \mathcal{L} . Another example is *Sharing* [14]. In fact the element $\{\emptyset, \{X\}, \{Y, Z\}\}$ can be represented as $\alpha_{=0}(X \cap Y) \wedge \alpha_{=0}(X \cap Z) \wedge \alpha_{=0}(Y \cap \overline{Z}) \wedge \alpha_{=0}(Z \cap \overline{Y})$. We recall that cardinality constraints are equipped with an operation of conjunction and cylindrification (existential quantifier).

We are thinking about ways of augmenting the expressive power of the logic \mathcal{L} , obviously while retaining the decidability. We are currently investigating two possibilities.

The first consists in adding a modal operator \Box defined as follows

$$\models_{\sigma} \Box \varphi \quad \text{iff for each } \sigma' \geq \sigma \models_{\sigma'} \varphi.$$

Such modality would allow, first of all, to characterize monotone properties of language \mathcal{L} . In fact monotone properties would correspond to the formulas Ψ such that $\Psi \Leftrightarrow \Box \Psi$. Moreover we could express arbitrary dependences between properties, like $\Box(list(X) \Rightarrow list(Y))$, whose informal meaning is that every state that instantiate X to a *list* will also bind Y to a *list*.

Another extension is to consider Hoare-like triples $\{\Phi\} \llbracket t, s \rrbracket \{\Psi\}$, whose meaning is: if Φ is true under the state σ and $\theta = mgu(\sigma(t), \sigma(s))$, then Ψ is true under the state $\sigma \circ \theta$. They have been considered in [7] and [8], where a formal calculus has been provided for a language of assertions different from \mathcal{L} . These formulas would allow to express formally the verification of general proof methods, like those proposed in [13] and in [16], for the whole class of formulas of \mathcal{L} . At the moment, it is still not known if, given a decidable logic such as \mathcal{L} , it is possible to decide the validity of such triples.

References

1. A. Bossi and N. Cocco. Verifying Correctness of Logic Programs. In J. Diaz and F. Orejas, editors, *Proc. TAPSOFT'89*, pages 96–110, 1989.
2. J. Boye. *Directional Types in Logic Programming*. PhD thesis, University of Linköping, Computer Science Department, 1997.
3. J. Boye and J. Maluszynski. Directional Types and the Annotation Method. *Journal of Logic Programming*, 33(3):179–220, 1997.
4. D. Cantone, E. G. Omodeo, and A. Policriti. The Automation of Syllogistic II. Optimization and Complexity Issues. *Journal of Automated Reasoning*, 6(2):173–187, 1990.
5. C. C. Chang and H. J. Kreisler. *Model Theory*. Elsevier Science Publ., 1990. Third edition.
6. K. L. Clark. Predicate logic as a computational formalism. Res. Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979.
7. L. Colussi and E. Marchiori. Proving Correctness of Logic Programs Using Axiomatic Semantics. In *Proc. of the Eight International Conference on Logic Programming*, pages 629–644. The MIT Press, Cambridge, Mass., 1991.
8. L. Colussi and E. Marchiori. Unification as Predicate Transformer. In *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 67–85. The MIT Press, Cambridge, Mass., 1992.
9. A. Cortesi, G. Filè, and W. Winsborough. Prop revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 322–327. IEEE Computer Society Press, 1991.
10. P. Dart and J. Zobel. Efficient run-time type checking of typed logic program. *Journal of Logic Programming*, 14(1-2):31–70, 1992.
11. P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in logic programming*, pages 157–187. The MIT Press, Cambridge, Mass., 1992.
12. P. Deransart. Proof Methods of Declarative Properties of Definite Programs. *Theoretical Computer Science*, 118(2):99–166, 1993.
13. W. Drabent and J. Maluszynski. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59(1):133–155, 1988.
14. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In E. Lusk and R. Overbeek, editors, *Proc. North American Conf. on Logic Programming'89*, pages 154–165. The MIT Press, Cambridge, Mass., 1989.
15. G. Kreisel and J. L. Krivine. *Elements of Mathematical Logic (Model Theory)*. North-Holland, Amsterdam, 1967.
16. G. Levi and P. Volpe. Derivation of Proof Methods by Abstract Interpretation. (Submitted). Available at <http://www.di.unipi.it/~volpe/papers.html>, 1998.
17. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.
18. E. Marchiori. A Logic for Variable Aliasing in Logic Programs. In G. Levi and M. Rodriguez-Artalejo, editors, *Proceedings of the 4th International Conference on Algebraic and Logic Programming (ALP'94)*, number 850 in LNCS, pages 287–304. Springer Verlag, 1994.
19. E. Marchiori. Design of Abstract Domains using First-order Logic. In M. Hanus and M. Rodriguez-Artalejo, editors, *Proceedings of the 5th International Conference on Algebraic and Logic Programming (ALP'96)*, number 1139 in LNCS, pages 209–223. Springer Verlag, 1996.

Refining Static Analyses by Trace-Based Partitioning Using Control Flow

Maria Handjieva and Stanislav Tzolovski

LIX, Ecole Polytechnique, France
{handjiev,stivy}@lix.polytechnique.fr

Abstract. This paper presents a systematic method of building a more precise static analysis from a given one. The key idea is to lift an abstract domain to the finite sets of its labeled abstract properties. The labels are designed to gather information about the history of control flow and to obtain a finite partitioning of the program execution traces. The abstract operations of the lifted domain are derived from those of the original one. This is a particular instance of the reduced cardinal power introduced by P. and R. Cousot, where the base is the set of labels approximating the control history and the exponent is an abstract domain. The method is applied to the domain of convex polyhedra and to the domain of linear congruences.

Key words: abstract interpretation, reduced cardinal power, trace semantics.

1 Introduction

An essential part of static analysis by abstract interpretation [45] is to build a machine-representable abstract domain expressing interesting properties about a computer program. Many abstract domains have been developed for many purposes [3,7,9,11]. The most significant analyses dealing with numerical variables are constant propagation, analysis using intervals, linear equalities, linear inequalities, linear congruences. We are interested in refining static analyses based on a given domain.

For non-distributive abstract interpretations (such as convex polyhedra), the disjunctive completion [6] is an operator that systematically produces new and more precise abstract domains from simpler ones. This was illustrated in [8] for the domain *PROP* for ground-dependence analysis of logic programs. We propose a method for lifting an abstract domain to the finite sets of its labeled abstract properties. The labels are designed to gather information about the history of control flow and to obtain a finite partitioning of the execution traces. Each partition (the abstract properties with the same label) is approximated with an upper bound of its elements. The abstract operations of the new domain are derived from those of the original one. We apply this method to the domain of convex polyhedra and to the domain of linear congruences.

A Motivated Example. A static analysis of program variables using convex polyhedra is proposed in [7]. We slightly change the illustrated example from [7] as follows:

```

{U0}   [c1] i = 2;   [c2] j = 0;
{U1}   [c3] if (Cond1) then
{U2}                               [c4] k = 1;
{U3}           else
{U4}                               [c5] k = -1;
{U5}           endif
{U6}   [c6] while (Cond2) do
{U7}   [c7]   if (Cond3) then
{U8}                               [c8] i = i + 4 * k;
{U9}           else
{U10}                              [c9] i = i + 2 * k; [c10] j = j + k;
{U11}          endif
{U12}          enddo;
{U13}   [c11] exit;
```

For each statement *Stat* of the program we have two abstract properties: U_i and U_o called input (before execution of *Stat*) and output (after execution of *Stat*) abstract properties respectively. We denote the label of statement *i* as $[c_i]$ and write the label before the statement.

If we apply the analysis based on convex polyhedra to the above program we will obtain the convex polyhedron $U_{13} = (-1 \leq k \leq 1)$. The result is too abstract, because the analysis cannot find that the value of variable *k* in the loop is either 1 or -1. The intuition is to represent the restraints between variables with two convex polyhedra (one for $k = -1$ and another for $k = 1$). In this case the result is $U_{13} = \{(i - 2j \leq 2, j \leq 0, k = -1), (i - 2j \geq 2, j \geq 0, k = 1)\}$, which is obviously more precise than the previously obtained one. Observe in this example, that this is the *reduced cardinal power* [5] with boolean base and polyhedral exponent. Therefore we need an abstract domain, which treats finite sets of convex polyhedra. Because the domain is not finite and does not satisfy the ascending chain condition we need also a technique that guarantees the termination of the analysis. Widening operators are an appropriate technique for this purpose. Moreover, we can use the same idea for other abstract domain, hence a method that systematically lifts a given abstract domain to the set of abstract properties is needed.

The rest of this paper is structured as follows. In the next section we introduce the set of labeled abstract properties corresponding to a given abstract domain. Starting from a concrete trace semantics and passing through a collecting semantics we design an abstract semantics using a concretization function and a widening. In Sects. [3] and [4] we apply our method to the convex polyhedra domain and to the domain of linear congruences respectively. We conclude the paper in Sect. [5].

2 Trace-Based Partitioning Using Control Flow

In order to simplify the presentation all formal reasoning is done using a simple programming language with assignments, “if-then-else” constructs and “while” loops. Our framework takes two parameters. The first one is an abstract domain:

$$(\mathcal{D}, \sqsubseteq_D, \sqcap_D, \sqcup_D, \perp_D, \top_D, \alpha_D, \gamma_D, \nabla_D, \text{assign}_D, \text{True}_D, \text{False}_D)$$

where \mathcal{D} is the set of abstract properties, \sqsubseteq_D is the abstract ordering, \sqcap_D and \sqcup_D are the upper and lower bound respectively, \perp_D and \top_D are the minimal and maximal element respectively, α_D is the abstraction function (if any) and γ_D is the concretization function, ∇_D is a widening operator (if any). We add three specific functions: $\text{assign}_D \in (\mathcal{D} \times \text{ASSIGN}) \rightarrow \mathcal{D}$ that treat the assignments (from ASSIGN), $\text{True}_D \in (\mathcal{D} \times \text{COND}) \rightarrow \mathcal{D}$ and $\text{False}_D \in (\mathcal{D} \times \text{COND}) \rightarrow \mathcal{D}$ that treat the conditions in the program. For example, in the domain of closed intervals (denoted by I) if the value of variable x is approximated with the interval $[1, 5]$ then:

$$\begin{aligned} \text{assign}_I([1, 5], x = x + 1) &= [2, 6] \\ \text{True}_I([1, 5], x \geq 3) &= [3, 5], \quad \text{False}_I([1, 5], x \geq 3) = [1, 3] \end{aligned}$$

An “if-then-else” or a “while” statement is called a *test node*. Each test node has a unique number. Let T be the set of test node numbers and t be the number of test nodes ($t = |T|$). Each test node can be analyzed in different ways. The analysis of an “if” statement could return a set of one abstract property, which contains information about both the “true” and “false” branches or it could return a set of two abstract properties one for each branch. In the case of a “while” statement the analysis could obtain a set of $k + 1$ elements: one - when the control does not pass through the loop; one - when the control passes exactly once through the loop, etc.; one - when the control passes k or more times through the loop. So, the second parameter of our framework is a tuple of t elements, denoted by Δ and called *test approximation parameter*, that contains information on how to approximate each test node.

2.1 Concrete Semantics

The concrete semantics of a program P at point z is the set of all execution traces from the entry point to point z .

$$\mathcal{S}_z[P] = \{[c_1, M_1] \rightarrow [c_2, M_2] \rightarrow \dots \rightarrow [c_z, M_z], \dots\}$$

where c_i is the label of statement i and M_i is the memory state (a vector of values) before the execution of statement i .

2.2 Collecting Semantics

The starting point of an abstract interpretation is a collecting semantics for the programming language. Our collecting semantics is a set of configurations. Each

configuration is a pair of a string of labels and a vector of values. In what follows configurations are always surrounded with left and right angles. For each trace in the concrete semantics $\mathcal{S}_z \llbracket P \rrbracket$ there is a configuration in the collecting semantics that characterizes the control history of this trace and its current memory state.

$$\mathcal{C}_z \llbracket P \rrbracket = \{ \langle c_1 c_2 \dots c_z, M_z \rangle, \langle c'_1 c'_2 \dots c'_z, M'_z \rangle, \dots, \langle c''_1 \dots c''_z, M''_z \rangle \}$$

The concretization γ_{cs} and abstraction α_{sc} functions, that describe the link between concrete and collecting semantics, are defined recursively as follows:

$$\begin{aligned} \gamma_{cs}(C) &= \bigcup_{\langle c_1 \dots c_z, M_z \rangle \in C} \gamma_t(\langle c_1 \dots c_z, M_z \rangle) \\ \gamma_t(\langle c_1 \dots c_z, M_z \rangle) &= [c_1, M_1] \rightarrow \gamma_t(\langle c_2 \dots c_z, M_z \rangle), \quad M_1 \text{ is any memory state} \\ \gamma_t(\langle c_z, M_z \rangle) &= [c_z, M_z] \end{aligned}$$

To define the abstraction function we use the following notations: all letters in bold are strings, ϵ is the empty string and “.” is the string concatenation.

$$\begin{aligned} \alpha_{sc}(T) &= \{ \alpha_t(\epsilon, t) \mid t \in T \} \\ \alpha_t(\mathbf{c}, [c, M] \rightarrow t) &= \alpha_t(\mathbf{c} \cdot c, t) \\ \alpha_t(\mathbf{c}, [c, M]) &= \langle \mathbf{c} \cdot c, M \rangle \end{aligned}$$

Note that the pair $(\alpha_{sc}, \gamma_{cs})$ is a Galois connection, since $\alpha_{sc}(\bigcup_i T_i) = \bigcup_i \alpha_{sc}(T_i)$.

2.3 Set of Labeled Abstract Property Domain

The second step is the abstraction of the collecting semantics. Our abstract domain is a finite set of labeled abstract properties from a given abstract domain.

An abstract label is a string from the alphabet $A = \{b, t_j, f_j, v_j\}$, for $j \in T$, such that it begins with b and there are no more b . The letter b describes the path from the entry node to the first test node. The letter t_j (resp. f_j) describes the part of the “true” (resp. “false”) branch of test node j (from j to the next test node). The abstract letter v_j represents:

- all paths that pass a finite number of times through j , if j is a “while” loop;
- the two paths that pass through j , if j is an “if-then-else” statement.

The concrete control flow graph contains one node for each statement and a node for each junction point (“endif”). The abstract control flow graph contains: one node for each test point j , labeled with j , an entry, an exit node and one node for each junction point. The junction nodes are added for simplification.

For example, the concrete control flow graph and the abstract control flow graph of the program from Sect. 1 are illustrated in Fig. [11](#).

where X and Y are regular expressions. So $\gamma_l(v_j)$ is $\gamma_l(\delta(G_j))$, where G_j is the subgraph of the abstract control flow graph with nodes from the strongly connected components with head j .

Function γ_L gives the meaning of an abstract label \mathbf{a} (which is a string of abstract letters $\mathbf{a} = a_1 \dots a_n$) :

$$\gamma_L(\mathbf{a}) = \gamma_L(a_1 \dots a_n) = \{\mathbf{s}_1 \dots \mathbf{s}_n \mid \mathbf{s}_i \in \gamma_l(a_i), i = 1, \dots, n\}$$

The abstract labels are ordered by:

$$\mathbf{a} \preceq \mathbf{a}' \text{ iff } \gamma_L(\mathbf{a}) \subseteq \gamma_L(\mathbf{a}')$$

For example, $bt_1t_2t_3 \preceq bt_1v_2$, but $bt_1f_2 \not\preceq bt_1v_2$.

An abstract configuration is an element of the reduced cardinal power [5] with sets of abstract labels as atomic base and abstract domain \mathcal{D} as exponent which represent as a pair of an abstract label and an abstract property, denoted as $\langle \mathbf{a}, D \rangle$, where $D \in \mathcal{D}$. Our abstract domain consists of sets of abstract configurations with an additional condition: there are not two elements with comparable labels in each set. This additional condition guarantees correct trace partitioning.

2.4 Abstract Operations

The last step consists in computing the least fixpoint of the abstract semantics. In this subsection we define the operations on the abstract domain needed when computing this least fixpoint (described in the next subsection).

Approximation ordering. We introduce the following ordering so that we can compare two sets of labeled abstract properties.

$$U \sqsubseteq U' \text{ iff } \forall \langle \mathbf{a}, D \rangle \in U, \exists \langle \mathbf{a}', D' \rangle \in U', \mathbf{a} \preceq \mathbf{a}' \wedge D \sqsubseteq_D D'$$

Upper bound. The intuition behind designing the upper bound operation is to obtain either a set union over the paths with non-comparable labels or merge over the paths with comparable labels. In this way we cannot merge paths from different partitioning.

$$\begin{aligned} U \sqcup U' = & \left\{ \langle \mathbf{a}, D \sqcup_D D'_1 \sqcup_D \dots \sqcup_D D'_n \rangle \mid \langle \mathbf{a}, D \rangle \in U, \langle \mathbf{a}'_k, D'_k \rangle \in U', \right. \\ & \left. \mathbf{a}'_k \preceq \mathbf{a}, k = 1, \dots, n \right\} \\ & \cup \left\{ \langle \mathbf{a}', D' \sqcup_D D_1 \sqcup_D \dots \sqcup_D D_m \rangle \mid \langle \mathbf{a}', D' \rangle \in U', \langle \mathbf{a}_k, D_k \rangle \in U, \right. \\ & \left. \mathbf{a}_k \preceq \mathbf{a}', k = 1, \dots, m \right\} \\ & \cup \{ \langle \mathbf{a}, D \rangle \mid \langle \mathbf{a}, D \rangle \in U, \nexists \langle \mathbf{a}', D' \rangle \in U', (\mathbf{a} \preceq \mathbf{a}' \vee \mathbf{a}' \preceq \mathbf{a}) \} \\ & \cup \{ \langle \mathbf{a}', D' \rangle \mid \langle \mathbf{a}', D' \rangle \in U', \nexists \langle \mathbf{a}, D \rangle \in U, (\mathbf{a} \preceq \mathbf{a}' \vee \mathbf{a}' \preceq \mathbf{a}) \} \end{aligned}$$

Widening operator. A convergence technique has been proposed in [4] that uses so called widening operator to transform infinitely iterative computation into finite but approximate one. Let $(L, \sqsubseteq, \perp, \top, \sqcup)$ be a poset with maximal element \top . The following definition is due to Cousot [2].

Definition 1 A widening operator $\nabla \in (\mathbf{N} \rightarrow (L \times L \rightarrow L))$ is such that¹:

1. $\forall i > 0, \forall x, y \in L, (x \sqcup y) \sqsubseteq (x \nabla_i y)$;
2. For any ascending chain $y^0 \sqsubseteq y^1 \sqsubseteq \dots \sqsubseteq y^n \sqsubseteq \dots$ of elements of L , the ascending chain $x^0 = y^0, x^1 = x^0 \nabla_1 y^1, \dots, x^n = x^{n-1} \nabla_n y^n, \dots$ is eventually stable, i.e. there exists $k \geq 0$ such that for $i > k, x^i = x^k$.

The widening on our abstract domain is defined as follows:

$$U \nabla U' = \{ \langle \mathbf{a}, D \nabla_D (D \sqcup_D D_1 \sqcup_D \dots \sqcup_D D_n) \rangle \mid \langle \mathbf{a}, D \rangle \in U, \langle \mathbf{a}_i, D_i \rangle \in U', \mathbf{a}_i \preceq \mathbf{a} \} \\ \cup \{ \langle \mathbf{a}, D \rangle \mid \langle \mathbf{a}, D \rangle \in U \wedge \nexists \langle \mathbf{a}', D' \rangle \in U', \mathbf{a}' \preceq \mathbf{a} \}$$

It is designed only for sets of labeled abstract properties U and U' such that:

$$\forall \langle \mathbf{a}', D' \rangle \in U', \exists \langle \mathbf{a}, D \rangle \in U, \mathbf{a}' \preceq \mathbf{a}$$

This condition is not a restriction because, the widening operator ∇ is always applied between two successive iterations of a “while” loop and we analyze only structured programs.

Proposition 1 The operator ∇ defined above is a widening.

Proof: It is clear that $U \sqcup U' \sqsubseteq U \nabla U'$. On the other hand, the number of abstract properties in $U \nabla U'$ is equal to the number of abstract properties in U ($|U| = |U \nabla U'|$). Moreover for each abstract label \mathbf{a} we cannot have infinite many abstract properties $D_1 \sqsubseteq_D D_2 \sqsubseteq_D \dots$ from successive iterations, because of the application of the widening ∇_D . Therefore the operator ∇ cannot iterate infinitely without convergence. \square

2.5 Abstract Semantics

The abstract semantics of a program P at point z is a set of abstract configurations $\langle \mathbf{a}, D \rangle$, where \mathbf{a} is an abstract label and $D \in \mathcal{D}$ is an abstract property.

$$\mathcal{A}_z \llbracket P \rrbracket = \{ \langle \mathbf{a}_i, D_i \rangle \mid i = 1, \dots, n \}$$

In our approach, the link between the abstract and collecting semantics is given by a monotone concretization function Γ defined below.

$$\Gamma(U) = \bigcup_{\langle \mathbf{a}, D \rangle \in U} \gamma(\langle \mathbf{a}, D \rangle) \\ \gamma(\langle \mathbf{a}, D \rangle) = \{ \langle \mathbf{s}, M \rangle \mid \mathbf{s} \in \gamma_L(\mathbf{a}), M \in \gamma_D(D) \}$$

Assignments. Consider the following assignment:

$$\{U\} \quad x = Expr \quad \{U'\}$$

¹ We denote ∇_n the operator $\nabla(n)$, $n \in \mathbf{N}$.

Its output set of labeled abstract properties is given by:

$$U' = \text{assign}(U, x = \text{Expr}) = \{\langle \mathbf{a}, D' \rangle \mid \langle \mathbf{a}, D \rangle \in U, D' = \text{assign}_D(D, x = \text{Expr})\}$$

If-then-else constructs. We now give the abstract meaning of an “if-then-else” statement, where **BlockThen** and **BlockElse** are blocks of statements:

$$\begin{array}{lll} \{U\} & \text{if (Cond) then} & \\ \{U_1\} & \quad \text{BlockThen} & \{U_{11}\} \\ & \text{else} & \\ \{U_2\} & \quad \text{BlockElse} & \{U_{22}\} \\ & \text{endif} & \{U'\} \end{array}$$

There are two possible ways to approximate the information after “if-then-else” statement i . If $k_i = 1$ (k_i is the i -th element of the test approximation parameter) then we set $l_t = t_i$ and $l_f = f_i$, otherwise (when $k_i = 0$) we set $l_t = l_f = v_i$.

$$\begin{aligned} U_1 &= \{\langle \mathbf{a} \cdot l_t, D' \rangle \mid \langle \mathbf{a}, D \rangle \in U, D' = \text{True}_D(D, \text{Cond}), D' \neq \perp_D\} \\ U_2 &= \{\langle \mathbf{a} \cdot l_f, D' \rangle \mid \langle \mathbf{a}, D \rangle \in U, D' = \text{False}_D(D, \text{Cond}), D' \neq \perp_D\} \\ U' &= U_{11} \sqcup U_{22} \end{aligned}$$

Note that when $k_i = 1$ there is not a merge of the “true” and “false” paths, because the upper bound is simply the set union. Which means that there is no loss of information in this case. Contrarily to common choice of abstracting the merge of these two paths by an upper bound on the domain, which is an upper approximation for non-distributive abstract interpretation frameworks. For example, in the convex polyhedra domain, where the union of two polyhedra does not necessarily correspond to the convex polyhedron, it is approximated by the convex hull.

While loops. Consider the following loop statement:

$$\begin{array}{lll} \{U\} & \text{while (Cond) do} & \\ \{U_1\} & \quad \text{Block} & \{U_2\} \\ & \text{enddo} & \{U'\} \end{array}$$

In order to find the fixpoint of the abstract semantics we use additional sets of labeled abstract property V^i which is always valid before the i -th iteration.

$$\begin{aligned} V^i &= \begin{cases} U, & \text{when } i = 0 \\ U_2^i, & \text{when } 0 < i < k_j \\ \{\langle \mathbf{a} \cdot v_j, D \rangle \mid \langle \mathbf{a}, D \rangle \in U_2^{k_j}\}, & \text{when } i = k_j \\ V^{i-1} \nabla U_2^i, & \text{when } i > k_j \end{cases} \\ U_1^i &= \{\langle \mathbf{a} \cdot t_j, D' \rangle \mid \langle \mathbf{a}, D \rangle \in V^{i-1}, D' = \text{True}_D(D, \text{Cond}), D' \neq \perp_D\} \\ U' &= \bigcup_{i=0, k_j-1} \{\langle \mathbf{a} \cdot f_j, D' \rangle \mid \langle \mathbf{a}, D \rangle \in V^i, D' = \text{False}_D(D, \text{Cond}), D' \neq \perp_D\} \\ &\quad \cup \{\langle \mathbf{a} \cdot f_j, D' \rangle \mid \langle \mathbf{a}, D \rangle \in V^{k_j+y}, D' = \text{False}_D(D, \text{Cond}), D' \neq \perp_D\} \end{aligned}$$

where y is such that the local fixpoint (for the loop j) is reached, which means the smallest y such that $V^{k_j+y} = V^{k_j+y+1}$.

2.6 Complexity and Precision

The complexity of our analysis depends on its two parameters - the abstract domain and the test approximation parameter. For each program Q , having its test approximation parameter Δ_Q we can calculate the maximal number of labeled abstract properties in the abstract semantics (which is in fact the number of partitions). Let σ be a function which for a given abstract control flow subgraph (with exactly one entry and exactly one exit edge) calculates this number according to $\Delta_Q = (k_1, k_2, \dots, k_t)$. We recursively define σ as follows:

$$\sigma \left(\begin{array}{c} \text{graph with entry } t_j \text{ and exit } f_j \\ \text{node } P \end{array}, u \right) = u + \sigma(P, u) + \dots + \underbrace{\sigma(P, \dots, \sigma(P, u) \dots)}_{k_j} \quad \delta(\cdot, u) = u$$

$$\sigma \left(\begin{array}{c} \text{graph with entry } t_j \text{ and exit } f_j \\ \text{nodes } P \text{ and } R \end{array}, u \right) = \begin{cases} u, & \text{if } k_j = 0 \\ \sigma(P, u) + \sigma(R, u), & \text{if } k_j = 1 \end{cases} \quad \sigma \left(\begin{array}{c} \text{graph with entry } t_j \text{ and exit } f_j \\ \text{nodes } P \text{ and } R \end{array}, u \right) = \sigma(R, \sigma(P, u))$$

where j is a test node, P and R are part of the abstract control flow graph with exactly one entry edge and exactly one exit edge and u is a number of partitions at the entry edge.

Therefore the maximal number of elements in the abstract semantics at the end of program Q is $\sigma(Q, 1)$. Obviously the analysis based on the lifted abstract domain is more costly than the analysis based on the original one. For example, the maximal number of elements in the set of labeled abstract properties at point 13 of the program from Sect. 1 is $2^{k_1} \sum_{i=0}^{k_2} 2^{k_3 i}$, where $k_1, k_3 \in \{0, 1\}$ and $k_2 \in \mathbf{N}$.

The precision of our analysis also depends on the test approximation parameter. We can argue that our analysis based on the lifted domain provides more accurate results than or the same results as the analysis based on abstract domain \mathcal{D} .

Proposition 2 *The analysis based on finite sets of labeled abstract properties of \mathcal{D} is more precise than the analysis based on \mathcal{D} .*

In the worst case of our analysis, when $\Delta = [0, \dots, 0]$ the result is a set of one abstract property, which is exactly the same as the result of the static analysis based on \mathcal{D} .

We can design a family of static analyses changing the test approximation parameter as making different compromises between complexity and precision.

3 Finite Sets of Labeled Convex Polyhedra

3.1 Convex Polyhedra Domain

Static analysis of linear inequalities among variables of a program have been studied by P.Cousot and N.Halbwachs in [7]. To set up some notations we briefly

present the definitions related to convex polyhedra [12]. The *convex hull* of a set X of vectors is the smallest convex set containing X and it is denoted by $\text{convexHull}(X)$, so:

$$\begin{aligned} \text{convexHull}(X) = \{ \lambda_1 x_1 + \dots + \lambda_k x_k \mid k \geq 1; x_1, \dots, x_k \in X; \\ \lambda_1, \dots, \lambda_k \in \mathbf{R}; \lambda_1, \dots, \lambda_k \geq 0; \lambda_1 + \dots + \lambda_k = 1 \} \end{aligned}$$

A set P of vectors in \mathbf{R}^n is called a *convex polyhedron* if $P = \{x \mid Ax \leq b\}$ for some matrix A and vector b . In other words, the polyhedron is the intersection of finitely many affine half-spaces. An *affine half-space* is a set $\{x \mid ax \leq b\}$ for some nonzero row vector a and some number b . A set of vectors is a *convex polytope* if it is the convex hull of finitely many vectors. The convex hull of two convex polyhedra is the smallest convex set defined as follows:

$$\text{chull}(P_1, P_2) = \text{convexHull}(\{X \mid X \in P_1 \vee X \in P_2\})$$

Upper Bound. The upper bound of two convex polyhedra is their convex hull. So, we have: $P_1 \sqcup_p P_2 = \text{chull}(P_1, P_2)$ and $P_1 \sqcup_p \dots \sqcup_p P_n = \text{chull}(P_1, \dots, P_n)$.

Affine Transformation. The affine transformation τ_p transforms a polyhedron P into another polyhedron P' according to a given affine map $x \rightarrow Ax + b$. Polyhedron P' is specified as follows: $P' = \{x' \mid x \in P, x' = Ax + b\}$

Widening Operator. The widening operator of convex polyhedra ∇_p (defined in [7] and optimized in [10]) is based on the following heuristic: the widening of two polyhedra is a new polyhedron obtained by removing from the system of the first polyhedron all inequalities which are not satisfied by the second polyhedron.

3.2 Specific Functions

In the following, specific functions assign_p , True_p and False_p that deal with convex polyhedra are presented.

Assignments. Let $x = \text{Expr}$ be an assignment. For a given polyhedron P , function assign_p returns a polyhedron P' in the following manner:

(a) If Expr is nonlinear expression we make a conservative approximation, assuming that any value can be assigned to x . Therefore, variable x is eliminated from P by projecting P along the x dimension.

(b) If Expr is linear expression, i.e. $\text{Expr} \equiv (\sum_{i=1}^n a_i x_i + b)$ then function assign_p is an affine transformation defined by the affine map $x \rightarrow \sum_{i=1}^n a_i x_i + b$. In the case of invertible assignments $x_k = \sum_{i=1}^n a_i x_i + b$ (when $a_k \neq 0$) the system of restraints of each polyhedron can be computed directly [7]. In the case of non-invertible assignments (when $a_k = 0$) we first project polyhedron P along the x_k dimension and then add the restraint $x_k = \sum_{i=1}^n a_i x_i + b$.

Conditions. According to condition **Cond** we have the following three cases:

(a) condition **Cond** is nonlinear. We ignore the test by setting both functions True_p and False_p to behave as the identity function on convex polyhedra: $\text{True}_p(P, \text{Cond}) = \text{False}_p(P, \text{Cond}) = P$;

(b) condition **Cond** is a linear equality, i.e. $\text{Cond} \equiv (aX = b)$. Let H be the hyperplane $\{X \in R^n \mid aX = b\}$. If polyhedron P is included in H then

$True_p(P, \text{Cond}) = P$ and $False_p(P, \text{Cond}) = \perp_p$. Otherwise $True_p(P, \text{Cond}) = P \cap H$ and $False_p(P, \text{Cond}) = P$;

(c) the condition Cond is a linear inequality, i.e. $\text{Cond} \equiv (aX \leq b)$. Let H_1 be the closed half-space $\{X \in R^n \mid aX \leq b\}$ and let H_2 be the closed half-space $\{X \in R^n \mid aX \geq b\}$. In this case we set $True_p(P, \text{Cond}) = P \cap H_1$ and $False_p(P, \text{Cond}) = P \cap H_2$.

3.3 Example

We illustrate the application of the method described in Sect. 2 using convex polyhedra domain on the motivated example (see Sect. II). The program is simple enough to allow hand computation and simple geometrical representation. All tests involve nonlinear conditions, which are not taken into account by the analysis. The test approximation parameter is set up to be $\Delta = (1, 0, 1)$. Which means that the first test node (the “if” with label $[c_3]$) will be treated exactly (because $k_1 = 1$), the second test point (the “while” loop) will be approximated at the beginning (because $k_2 = 0$) etc. Each set of labeled polyhedra U_i , ($i = 0, \dots, 13$) is initially empty ($U_i^0 = \emptyset$). The superscript shows the number of steps the analysis passes through point i . At the entry node we set up $U_0^1 = \mathbf{R}^n$, where n is the number of variables involved in the analysis.

$$\begin{aligned}
 U_0^1 &= \mathbf{R}^3 \\
 U_1^1 &= \text{assign}(\text{assign}(U_0^1, i = 2), j = 0) = \{\langle b, (i = 2, j = 0) \rangle\} \\
 U_2^1 &= \{\langle bt_1, (i = 2, j = 0) \rangle\} \\
 U_3^1 &= \text{assign}(U_2^1, k = 1) = \{\langle bt_1, (i = 2, j = 0, k = 1) \rangle\} \\
 U_4^1 &= \{\langle bf_1, (i = 2, j = 0) \rangle\} \\
 U_5^1 &= \text{assign}(U_4^1, k = -1) = \{\langle bf_1, (i = 2, j = 0, k = -1) \rangle\} \\
 U_6^1 &= U_3^1 \sqcup U_5^1 = \{\langle bt_1, (i = 2, j = 0, k = 1) \rangle, \langle bf_1, (i = 2, j = 0, k = -1) \rangle\}
 \end{aligned}$$

Now, we are going to analyze the “while” loop.

$$\begin{aligned}
 V^0 &= \{\langle bt_1 v_2, (i = 2, j = 0, k = 1) \rangle, \langle bf_1 v_2, (i = 2, j = 0, k = -1) \rangle\} \\
 U_7^1 &= \{\langle bt_1 v_2 t_2, (i = 2, j = 0, k = 1) \rangle, \langle bf_1 v_2 t_2, (i = 2, j = 0, k = -1) \rangle\} \\
 U_8^1 &= \{\langle bt_1 v_2 t_2 t_3, (i = 2, j = 0, k = 1) \rangle, \langle bf_1 v_2 t_2 t_3, (i = 2, j = 0, k = -1) \rangle\} \\
 U_9^1 &= \text{assign}(U_8^1, i = i + 4 * k) = \\
 &= \{\langle bt_1 v_2 t_2 t_3, (i = 6, j = 0, k = 1) \rangle, \langle bf_1 v_2 t_2 t_3, (i = -2, j = 0, k = -1) \rangle\} \\
 U_{10}^1 &= \{\langle bt_1 v_2 t_2 f_3, (i = 2, j = 0, k = 1) \rangle, \langle bf_1 v_2 t_2 f_3, (i = 2, j = 0, k = -1) \rangle\} \\
 U_{11}^1 &= \text{assign}(\text{assign}(U_{10}^1, i = i + 2 * k), j = j + k) = \\
 &= \{\langle bt_1 v_2 t_2 f_3, (i = 4, j = 1, k = 1) \rangle, \langle bf_1 v_2 t_2 f_3, (i = 0, j = -1, k = -1) \rangle\} \\
 U_{12}^1 &= U_9^1 \sqcup U_{11}^1 = \\
 &= \left\{ \langle bt_1 v_2 t_2 t_3, (i = 6, j = 0, k = 1) \rangle, \langle bf_1 v_2 t_2 t_3, (i = -2, j = 0, k = -1) \rangle \right. \\
 &\quad \left. \langle bt_1 v_2 t_2 f_3, (i = 4, j = 1, k = 1) \rangle, \langle bf_1 v_2 t_2 f_3, (i = 0, j = -1, k = -1) \rangle \right\}
 \end{aligned}$$

If we unroll the loop once again we will obtain the following result:

$$\begin{aligned} V^1 &= V^0 \nabla U_{12}^1 = \\ &= \left\{ \langle bt_1v_2, (i - 2j \geq 2, i + 2j \leq 6, j \geq 0, k = 1) \rangle \right. \\ &\quad \left. \langle bf_1v_2, (i - 2j \leq 2, i + 2j \geq -2, j \leq 0, k = -1) \rangle \right\} \end{aligned}$$

The above widening behaves as an upper bound. It returns a set of two polyhedra. The first one is the convex hull of the polyhedra with labels bt_1v_2 , $bt_1v_2t_2t_3$ and $bt_1v_2t_2f_3$ (because $bt_1v_2t_2t_3 \preceq bt_1v_2$, $bt_1v_2t_2f_3 \preceq bt_1v_2$) and the second one is the convex hull of the polyhedra with labels bf_1v_2 , $bf_1v_2t_2t_3$ and $bf_1v_2t_2f_3$.

$$U_{12}^2 = \left\{ \begin{aligned} &\langle bt_1v_2t_2t_3, (i - 2j \geq 6, i + 2j \leq 10, j \geq 0, k = 1) \rangle \\ &\langle bf_1v_2t_2t_3, (i - 2j \leq -2, i + 2j \geq -6, j \leq 0, k = -1) \rangle \\ &\langle bt_1v_2t_2f_3, (i - 2j \geq 2, i + 2j \leq 10, j \geq 1, k = 1) \rangle \\ &\langle bf_1v_2t_2f_3, (i - 2j \leq 2, i + 2j \geq -6, j \leq -1, k = -1) \rangle \end{aligned} \right\}$$

At this point the widening operator takes place:

$$V^2 = V^1 \nabla U_{12}^2 = \left\{ \langle bt_1v_2, (i - 2j \geq 2, j \geq 0, k = 1) \rangle, \right. \\ \left. \langle bf_1v_2, (i - 2j \leq 2, j \leq 0, k = -1) \rangle \right\}$$

Finally, this is the fixpoint, therefore we find:

$$U_{13}^1 = \left\{ \langle bt_1v_2f_2, (i - 2j \geq 2, j \geq 0, k = 1) \rangle, \right. \\ \left. \langle bf_1v_2f_2, (i - 2j \leq 2, j \leq 0, k = -1) \rangle \right\}$$

Some of these results ($V^1, U_{13}^1 = V^2, U_{12}^2$) are shown on Fig. 2.

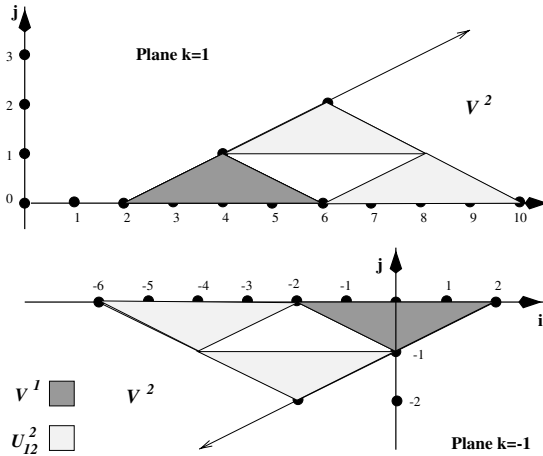


Fig. 2. The set of polyhedra for the motivated example

4 Finite Sets of Labeled Linear Congruences

4.1 Linear Congruence Domain

The static analysis of linear congruence equalities among program variables was studied by P.Granger in [9]. We briefly recall some notions related to linear congruences. Let G be an abelian group and H be a subgroup of G . Any subset of G of the form:

$$a + H = \{x \in G \mid \exists h \in H, x = a + h\}$$

where a is an element of G is called *coset* of H in G . The element a is said to be a *representative* of the coset and H the unique *modulo* of the coset. For any element b of $a + H$, we have $a + H = b + H$. The set of all cosets in the abelian group G with the singleton $\{\emptyset\}$ is called the *congruence lattice* of G and is denoted as $\mathcal{C}(G)$ (or \mathcal{C} when G is \mathbf{Z}). The set inclusion order can be simplified by the following formula:

$$(a_1 + H_1) \subseteq (a_2 + H_2) \equiv (a_1 - a_2 \in H_2) \wedge (H_1 \subseteq H_2)$$

Upper bound. The upper bound operations \sqcup in $\mathcal{C}(G)$ is

$$(a_1 + H_1) \sqcup (a_2 + H_2) = a_1 + \mathbf{Z}(a_1 - a_2) + H_1 + H_2$$

where $\mathbf{Z}a$ denotes the subgroup generated by a i.e. $\{\dots, -a - a, -a, 0, a, a + a, \dots\}$. In what follows we use \mathbf{Z}^n as a group G .

Affine transformation. We use $r(H)$ to denote the rank of H and $k(u)$ the set $\{X \in \mathbf{Z}^n \mid u(X) = (0, \dots, 0)\}$. Let τ_D be an affine transformation on \mathbf{Z}^n , u be its linear part and $a \in \mathbf{Z}^n$, $(e_i)_{1 \leq i \leq k}$ be a basis of the subgroup H of \mathbf{Z}^n . The coset $a + H$ is transformed by τ_D in the following manner:

$$\begin{aligned} \tau_D(a + H) &= \tau_D(a + \mathbf{Z}e_1 + \dots + \mathbf{Z}e_k) = \tau_D(a) + \mathbf{Z}u(e_1) + \dots + \mathbf{Z}(e_k) \\ &= \tau_D(a) + \mathbf{Z}u(H) \end{aligned}$$

where the rank of $u(H)$ is equal to $k - r(k(u) \cap (H))$.

We have to mention that the congruence lattice satisfies the ascending chain condition, that guarantees the systematic termination of the analysis. This can be deduced from the fact that the complete lattice of all subgroups of \mathbf{Z}^n satisfies the ascending chain condition. Therefore no widening operators are needed for this domain.

4.2 Lifting the Domain of Linear Congruences

The domain of sets of labeled congruences is obtained by lifting the domain of linear congruences using the method described in Sect. 2. The abstract semantics for an imperative programming language with assignments, “if-then-else” and “while” statements is close to this in Subsect. 2.5. Only the “while” statement is treated in a different manner, because no widening is needed for this analysis. Therefore, the widening operator is simply replaced by the upper bound operator.

4.3 Example

Let us now illustrate with help of the motivated example the analysis based on finite sets of labeled linear congruences. As in Subject. [3.3](#) $\Delta = (1, 0, 1)$. Before the while loop at point 6 of the program we have:

$$U_1^1 = \left\{ \left\langle b, \begin{pmatrix} 2 \\ 0 \end{pmatrix} \right\rangle \right\} \quad U_6^1 = \left\{ \left\langle bt_1, \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} \right\rangle, \left\langle bf_1, \begin{pmatrix} 2 \\ 0 \\ -1 \end{pmatrix} \right\rangle \right\}$$

The first pass of the analysis through the “while” loop gives:

$$U_{12}^1 = \left\{ \left\langle bt_1v_2t_2t_3, \begin{pmatrix} 6 \\ 0 \\ 1 \end{pmatrix} \right\rangle, \left\langle bf_1v_2t_2t_3, \begin{pmatrix} -2 \\ 0 \\ -1 \end{pmatrix} \right\rangle \right\} \\ \left\{ \left\langle bt_1v_2t_2f_3, \begin{pmatrix} 4 \\ 1 \\ 1 \end{pmatrix} \right\rangle, \left\langle bf_1v_2t_2f_3, \begin{pmatrix} 0 \\ -1 \\ -1 \end{pmatrix} \right\rangle \right\}$$

The second pass produces the following set of labeled congruences:

$$V^2 = V^1 \nabla U_{12}^1 = \left\{ \left\langle bt_1v_2, \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 4 & 2 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \mathbf{Z}^2 \right\rangle, \left\langle bf_1v_2, \begin{pmatrix} 2 \\ 0 \\ -1 \end{pmatrix} + \begin{pmatrix} 4 & 2 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \mathbf{Z}^2 \right\rangle \right\}$$

$$U_7^2 = U_8^2 = U_9^2 = U_{10}^2 = U_{11}^2 = U_{12}^2 = V^2$$

The analysis converges in the next iteration step (the third one), because $V^3 = V^2$. The interpretation of the obtained result is the two sets - $\{(i, j, k) \in \mathbf{Z}^3 \mid i - 2j \equiv 2 \pmod{4}, k \equiv 1 \pmod{0}\}$ and $\{(i, j, k) \in \mathbf{Z}^3 \mid i - 2j \equiv 2 \pmod{4}, k \equiv -1 \pmod{0}\}$. This is a more precise result than the following one obtained by the analysis based on linear congruences $\{(i, j, k) \in \mathbf{Z}^3 \mid i - 2j \equiv 2 \pmod{4}, k \equiv 1 \pmod{2}\}$.

5 Conclusions and Further Work

We have shown how to lift an abstract domain to the finite sets of its labeled abstract properties using control flow information. The significance of this method is that it always produces a static analysis which performs as well as or better than the original one. We have shown how this method can be used with two currently available abstract domains - convex polyhedra and linear congruences. The ideas are in fact general enough to capture other abstract properties as interval congruences, congruence properties on rational numbers etc.

The main contributions of this work are:

- the design of a method that lifts an abstract domain to the finite sets of its labeled abstract properties as a particular instance of the reduced cardinal power [\[5\]](#), where the base is the set of labels approximating the control history and the exponent is the original domain;

- the application of this method to the convex polyhedra domain and to the domain of linear congruences;

An implementation is currently ongoing and it will allow us to investigate how the algorithms perform in practice.

Acknowledgements. We would like to thank Patrick Cousot for pointing out the connection with the reduced cardinal power. We are grateful to our colleagues from LIX for helpful discussions.

References

1. F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4) (1992) 407-435.
2. P. Cousot. Semantic Foundations of Program Analysis. In Muchnick and Jones Eds. *Program Flow Analysis, Theory and Applications*, pp. 303-343, Prentice-Hall, 1981.
3. P. Cousot and R. Cousot. Static determination of dynamic properties of programs, In *Proceedings of the 2nd Int. Symposium on Programming*, pp. 106-130, 1976.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pp. 238-252, 1977.
5. P. Cousot, and R. Cousot, Systematic Design of Program Analysis frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pp. 269-282, 1979.
6. P. Cousot, and R. Cousot, Abstract Interpretation and Application to Logic Programs. In *Journal of Logic Programming*, pp. 103-179, 1992.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pp. 84-97, 1978.
8. G. Filé and F. Ranzato, Improving abstract interpretations by systematic lifting to the powerset. In *Proceedings of the International Logic Programming Symposium*, Ithaca, NY, pages 655-669. The MIT Press, 1994.
9. P. Granger, Static analysis of linear congruence equalities among variables of a program, In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, pp. 169-192, number 493 in LNCS, 1991.
10. N. Halbwachs and Y.-E. Proy and P. Raymond, Verification of linear hybrid systems by means of convex approximations, In *Proceedings of the International Static Analysis Symposium*, pp. 223-237, number 864 in LNCS, 1994.
11. F. Masdupuy. Array operations abstractions using semantics analysis of trapezoid congruences. In *Proceedings of the International Conference on Supercomputing*, Washington, 1992.
12. A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley & Sons, 1986.

Building Complete Abstract Interpretations in a Linear Logic-Based Setting

Roberto Giacobazzi[†] Francesco Ranzato^{‡,*} Francesca Scozzari[†]

[†]*Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
{giaco,scozzari}@di.unipi.it*

[‡]*Dipartimento di Matematica Pura ed Applicata, Università di Padova
Via Belzoni 7, 35131 Padova, Italy
franz@math.unipd.it*

Abstract. Completeness is an important, but rather uncommon, property of abstract interpretations, ensuring that abstract computations are as precise as possible w.r.t. concrete ones. It turns out that completeness for an abstract interpretation depends only on its underlying abstract domains, and therefore it is an abstract domain property. Recently, the first two authors proved that for a given abstract domain A , in all significant cases, there exists the most abstract domain, called least complete extension of A , which includes A and induces a complete abstract interpretation. In addition to the standard formulation, we introduce and study a novel and particularly interesting type of completeness, called observation completeness. Standard and observation completeness are here considered in the context of quantales, i.e. models of linear logic, as concrete interpretations. In this setting, we prove that various kinds of least complete and observationally complete extensions exist and, more importantly, we show that such complete extensions can all be explicitly characterized by elegant linear logic-based formulations. As an application, we determine the least complete extension of a generic abstract domain w.r.t. a standard bottom-up semantics for logic programs observing computed answer substitutions. This general result is then instantiated to the relevant case of groundness analysis.

1 Introduction

It is widely held that the ideal goal of any semantics design method is to find sound and complete representations for some properties of concrete (actual) computations. Abstract interpretation is one such methodology, where soundness is always required, while completeness more rarely holds. Completeness issues in abstract interpretation have been studied since the Cousot and Cousot seminal paper [5]. The intuition is that a complete abstract interpretation induces an

* The work of Francesco Ranzato has been supported by an individual post-doctoral grant from Università di Padova, Italy.

abstract semantics which is as precise as possible relatively to its underlying abstract domains and to the concrete interpretation of reference. The paradigmatic rule of signs is a typical and simple example of an abstract interpretation which is (sound and) complete for integer multiplication but merely sound for integer addition [13].

Although in static program analysis decidability issues often force to sacrifice completeness for achieving termination and/or efficiency, examples of complete abstract interpretations are common in other fields of application. For instance, several complete abstractions of algebraic polynomial systems have been studied by Cousot and Cousot in [7] and many complete abstract interpretations can be found in comparative program semantics [36,9]. Moreover, being completeness a notion relative to the concrete semantics of reference, complete abstract interpretations which are more concrete than a certain, possibly approximated and decidable, property of interest, yield an absolute upper bound for the precision that one can achieve in computing that property. Thus, complete abstract interpretations may play a useful rôle in static program analysis as well. These argumentations probably stimulated the recent trend of research on completeness in abstract interpretation [2,10,11,13,15,17,18].

One key feature of completeness in abstract interpretation is that this property uniquely depends upon the abstraction function. Let us denote by \mathcal{L}_C the so-called lattice of abstract interpretations of a concrete domain C (cf. [4,5]), where, for all $A, B \in \mathcal{L}_C$, $A \sqsubseteq B$ means that A is more precise (i.e. concrete) than B . Let us consider the simple case of an abstract interpretation $f^\# : A \rightarrow A$ of a concrete semantic function $f : C \rightarrow C$, where the abstract domain $A \in \mathcal{L}_C$ is related to C by an adjoint pair of abstraction and concretization maps $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$. Then, $f^\#$ is (sound and) complete if $\alpha \circ f = f^\# \circ \alpha$. It is easily seen that if $f^\#$ is complete then the best correct approximation f^b of f in A , i.e. $f^b \stackrel{\text{def}}{=} \alpha \circ f \circ \gamma : A \rightarrow A$, is complete as well, and, in this case, $f^\#$ indeed coincides with f^b (cf. [10]). Thus, given an abstract domain $A \in \mathcal{L}_C$, one can define a complete abstract semantic function $f^\# : A \rightarrow A$ over A if and only if $f^b : A \rightarrow A$ is complete. This simple observation makes completeness an abstract domain property, namely a characteristic of the abstract domain. It is then clear that a key problem consists in devising systematic and constructive methodologies for transforming abstract domains in such a way that completeness is achieved and the resulting complete abstract domains are as close as possible to the initial (noncomplete) ones. This problem has been first raised in a predicate-based approach to abstract interpretation by Mycroft [13, Section 3.2], who gave a methodology for deriving the most concrete domain, called “canonical abstract interpretation”, which is complete and included in a given domain of properties. More recently, the first two authors proved in [10] that when the concrete semantic function f is continuous, any domain A can always be extended into the most abstract domain which includes A and is complete for f — the so-called *least complete extension* of A . Analogously, [10] solved the aforementioned problem raised by Mycroft, by showing that, for any given domain A , the most concrete domain which is complete and included in A — the so-called *complete kernel*

of A — exists for any monotone semantic function. Very recently, we improved considerably such results, by providing explicit constructive characterizations for least complete extensions and complete kernels of abstract domains [11].

In this paper, we are concerned with completeness problems arising when concrete semantic binary¹ operations of type $C_1 \times C_2 \rightarrow C$ are assumed to give rise to a generalized form of quantales, called *typed quantales*. Quantales are well-known algebraic structures which turn out to be models of intuitionistic linear logic [16,19]. In this logical setting, we provide elegant linear logic-based solutions to a number of interesting completeness problems for abstract interpretations. Such solutions find relevant applications in static program analysis and comparative semantics, for instance in logic programming, where unification — the prime computational step of any logic program semantics — turns out to be a binary operation in a quantale of substitutions, or in data structure analysis, considering binary data constructors such as *cons* for lists. More in detail, a typed quantale $\langle C, C_1, C_2, \otimes \rangle$ consists of three complete lattices C , C_1 and C_2 , and of an operation $\otimes : C_1 \times C_2 \rightarrow C$ which is additive (i.e., preserves lub's) on both arguments. When $C = C_1 = C_2$, typed quantales boil down to standard quantales $\langle C, \otimes \rangle$. The main feature of (typed) quantales is that they support a notion of left and right linear implication between domain's objects: Given $a \in C_1$ and $b \in C$, there exists a unique greatest object $a \rightarrow b \in C_2$ which, when combined by \otimes with a , gives a result less than or equal to b . In other terms, the following right modus ponens law $a \otimes x \leq b \Leftrightarrow x \leq a \rightarrow b$ holds. Analogous left implicational objects exist for a corresponding left modus ponens law.

When solving completeness problems in a setting where concrete interpretations are typed quantales, implicational domain objects allow to elegantly characterize complete abstract domains in a variety of situations. Our first result provides a characterization based on linear implications between domain's objects of the least complete extension of any abstract domain of any quantale. Then, we consider the following completeness problem over typed quantales: Given a typed quantale $\langle C, C_1, C_2, \otimes \rangle$, a fixed abstraction $A \in \mathcal{L}_C$, with corresponding abstraction map $\alpha_A : C \rightarrow A$, and a pair of abstract domains $\langle A_1, A_2 \rangle \in \mathcal{L}_{C_1} \times \mathcal{L}_{C_2}$, does there exist the most abstract pair of domains $\langle A'_1, A'_2 \rangle \in \mathcal{L}_{C_1} \times \mathcal{L}_{C_2}$, with corresponding abstraction maps $\alpha_{A_i} : C_i \rightarrow A_i$ ($i = 1, 2$), such that $\langle A'_1, A'_2 \rangle \sqsubseteq_{\mathcal{L}_{C_1} \times \mathcal{L}_{C_2}} \langle A_1, A_2 \rangle$ and $\alpha_A(_ \otimes _) = \alpha_A(\alpha_{A'_1}(_) \otimes \alpha_{A'_2}(_))$? Here, the *observation* domain A is fixed, and we are thus looking for the most abstract pair of domains in $\mathcal{L}_{C_1} \times \mathcal{L}_{C_2}$ which is more concrete than an initial pair $\langle A_1, A_2 \rangle$ and simultaneously induces a complete abstract interpretation w.r.t. \otimes . This is termed an *observation completeness* problem. Again, solutions to this observation completeness problem are built in terms of linear implications between domains.

To illustrate the practical scope of our results, we first consider a simple example in data structure analysis involving abstract domains for lists. In particular, our results are applied in order to solve various observation completeness problems concerning abstract domains useful for detecting irredundant lists of

¹ Clearly, a generalization to n -ary operations would be straightforward.

objects. Then, in the context of logic program semantics, we consider an immediate consequences operator T_P defined in terms of unification and union of sets of idempotent substitutions, and characterizing computed answer substitutions in a s-semantics style (cf. [18]). As usual, unification turns out to be the key operation to take into account in order to build least complete extensions of abstract domains. Sets of idempotent substitutions and unification give rise to a unital commutative quantale: Given an abstract domain A , we show how the least complete extension of A w.r.t. this quantale naturally induces the least complete extension of A w.r.t. T_P functions. This permits to give explicitly, in terms of linear implications, the least complete extension, for any T_P , of a generic domain abstracting sets of substitutions. As a remarkable instance of our construction, we characterize the least complete extension of the plain groundness domain w.r.t. computed answer substitutions s-semantics.

2 Basic Notions

The lattice of abstract interpretations. In standard Cousot and Cousot's abstract interpretation theory, abstract domains can be equivalently specified either by Galois connections (GCs), i.e. adjunctions, or by upper closure operators (uco's) [5]. In the first case, the concrete and abstract domains C and A (both assumed to be complete lattices) are related by a pair of adjoint functions of a GC (α, C, A, γ) . Also, it is generally assumed that (α, C, A, γ) is a Galois insertion (GI), i.e. α is onto or, equivalently, γ is 1-1. In the second case instead, an abstract domain is specified as a uco on the concrete domain C , i.e. a monotone, idempotent and extensive operator on C . These two approaches are equivalent, modulo isomorphic representation of domain's objects. Given a complete lattice C , it is well known that the set $uco(C)$ of all uco's on C , endowed with the point-wise ordering \sqsubseteq , is a complete lattice $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \lambda x. \top_C, id \rangle$ (id denotes the identity function). Let us also recall that each $\rho \in uco(C)$ is uniquely determined by the set of its fixpoints, which is its image, i.e. $\rho(C) = \{x \in C \mid \rho(x) = x\}$, and that $\rho \sqsubseteq \eta$ iff $\eta(C) \subseteq \rho(C)$. Moreover, a subset $X \subseteq C$ is the set of fixpoints of a uco on C iff X is meet-closed, i.e. $X = \bigwedge(X) \stackrel{\text{def}}{=} \{\bigwedge_C Y \mid Y \subseteq X\}$ (note that $\top_C = \bigwedge_C \emptyset \in \bigwedge(X)$). Often, we will identify closures with their sets of fixpoints. This does not give rise to ambiguity, since one can distinguish their use as functions or sets according to the context. In view of the equivalence above, throughout the paper, $\langle uco(C), \sqsubseteq \rangle$ will play the rôle of the lattice \mathcal{L}_C of abstract interpretations of C [4,5], i.e. the complete lattice of all possible abstract domains of the concrete domain C . For an abstract domain $A \in \mathcal{L}_C$, $\rho_A \in uco(C)$ will denote the corresponding uco on C , and if A is specified by a GI (α, C, A, γ) then $\rho_A \stackrel{\text{def}}{=} \gamma \circ \alpha$. The ordering on $uco(C)$ corresponds to the standard order used to compare abstract domains with regard to their precision: A_1 is more precise than A_2 (i.e., A_1 is more concrete than A_2 or A_2 is more abstract than A_1) iff $A_1 \sqsubseteq A_2$ in $uco(C)$. Lub and glb on $uco(C)$ have therefore the following reading as operators on domains. Suppose $\{A_i\}_{i \in I} \subseteq uco(C)$: (i) $\sqcup_{i \in I} A_i$ is the most concrete among the domains which are abstractions of all

the A_i 's, i.e. it is their least (w.r.t. \sqsubseteq) common abstraction; (ii) $\prod_{i \in I} A_i$ is the most abstract among the domains (abstracting C) which are more concrete than every A_i ; this domain is also known as reduced product of all the A_i 's.

Quantales and linear logic. Quantales originated as algebraic foundations of the so-called quantum logic. They have been successively considered for the lattice-theoretic semantics of Girard's linear logic (see [16] for an exhaustive treatment of quantales). We introduce a mild generalization of the notion of quantale, which, up to knowledge, appears to be new. A *typed quantale* is a multisorted algebra $\langle C, C_1, C_2, \otimes \rangle$, where C, C_1, C_2 are complete lattices and $\otimes : C_1 \times C_2 \rightarrow C$ is a function such that $(\bigvee_i x_i) \otimes c_2 = \bigvee_i (x_i \otimes c_2)$ and $c_1 \otimes (\bigvee_i x_i) = \bigvee_i (c_1 \otimes x_i)$. In other terms, a typed quantale is a 3-sorted algebra endowed with a "product" \otimes which distributes over arbitrary lub's on both sides. Thus, for any $c_1 \in C_1$ and $c_2 \in C_2$, both functions $c_1 \otimes _$ and $_ \otimes c_2$ have right adjoints denoted, resp., by $c_1 \rightarrow _$ and $_ \leftarrow c_2$. Hence, for all $c \in C$, $c_1 \otimes c_2 \leq c \Leftrightarrow c_2 \leq c_1 \rightarrow c$, and, dually, $c_1 \otimes c_2 \leq c \Leftrightarrow c_1 \leq c \leftarrow c_2$. Two functions $\rightarrow : C_1 \times C \rightarrow C_2$ and $\leftarrow : C \times C_2 \rightarrow C_1$ can be therefore defined as follows:

$$c_1 \rightarrow c \stackrel{\text{def}}{=} \bigvee \{z \in C_2 \mid c_1 \otimes z \leq c\}; \quad c \leftarrow c_2 \stackrel{\text{def}}{=} \bigvee \{y \in C_1 \mid y \otimes c_2 \leq c\}.$$

Any typed quantale $\langle C, C_1, C_2, \otimes \rangle$ enjoys the following main properties: For all $c \in C$, $\{x_i\}_{i \in I} \subseteq C$, $c_1 \in C_1$, $\{y_i\}_{i \in I} \subseteq C_1$, $c_2 \in C_2$ and $\{z_i\}_{i \in I} \subseteq C_2$:

$$\begin{array}{ll} \text{(i)} & c_1 \otimes (c_1 \rightarrow c) \leq c \\ \text{(ii)} & (c \leftarrow c_2) \otimes c_2 \leq c \\ \text{(iii)} & c_1 \rightarrow (\bigwedge_{i \in I} x_i) = \bigwedge_{i \in I} (c_1 \rightarrow x_i) \\ \text{(iv)} & (\bigwedge_{i \in I} x_i) \leftarrow c_2 = \bigwedge_{i \in I} (x_i \leftarrow c_2) \\ \text{(v)} & (\bigvee_{i \in I} y_i) \rightarrow c = \bigwedge_{i \in I} (y_i \rightarrow c) \\ \text{(vi)} & c \leftarrow (\bigvee_{i \in I} z_i) = \bigwedge_{i \in I} (c \leftarrow z_i) \end{array}$$

When $C = C_1 = C_2$ and \otimes is associative, a typed quantale is called *quantale*. It is well known that quantales turn out to be models of noncommutative intuitionistic linear logic [16, 19]. A quantale $\langle C, \otimes \rangle$ is called *commutative* when \otimes is commutative, and this is equivalent to require that, for all $a, b \in C$, $a \rightarrow b = b \leftarrow a$. Also, a commutative quantale $\langle C, \otimes \rangle$ is called *unital* if there exists an object $1 \in C$ such that $1 \otimes a = a = a \otimes 1$, for all $a \in C$. For a quantale $\langle C, \otimes \rangle$, the following additional properties hold for all $a, b, c \in C$:

$$\begin{array}{ll} \text{(vii)} & a \rightarrow (c \leftarrow b) = (a \rightarrow c) \leftarrow b \\ \text{(viii)} & b \rightarrow (a \rightarrow c) = (a \otimes b) \rightarrow c \\ \text{(ix)} & (c \leftarrow b) \leftarrow a = c \leftarrow (a \otimes b) \end{array}$$

3 Completeness Problems in Abstract Interpretations

Let $\langle C, C_1, C_2, \otimes \rangle$ be a concrete interpretation, i.e. C, C_1 and C_2 are concrete semantic domains provided with a semantic operation $\otimes : C_1 \times C_2 \rightarrow C$. When

$C = C_1 = C_2$, we adopt the simpler notation $\langle C, \otimes \rangle$. Given the abstractions $A_1 \in \mathcal{L}_{C_1}$, $A_2 \in \mathcal{L}_{C_2}$ and $A \in \mathcal{L}_C$, let us recall [5] that the best correct approximation $\otimes^b : A_1 \times A_2 \rightarrow A$ of \otimes is defined as $\otimes^b \stackrel{\text{def}}{=} \alpha_{C,A} \circ \otimes \circ \langle \gamma_{A_1,C_1}, \gamma_{A_2,C_2} \rangle$. It has been shown in [10] that completeness for an abstract interpretation is a property depending only on the underlying abstract domains. In our setting, this means that an abstract interpretation $\langle A, A_1, A_2, \otimes^\# \rangle$, with $\otimes^\# : A_1 \times A_2 \rightarrow A$, is complete for $\langle C, C_1, C_2, \otimes \rangle$ iff $\langle A, A_1, A_2, \otimes^b \rangle$ is complete and $\otimes^\# = \otimes^b$. In other terms, the best correct approximation induced by the underlying abstract domains determines the property of being complete. Hence, we find more convenient, elegant and, of course, completely equivalent, to reason on completeness by using abstract domains specified by the closure operator approach.

Full completeness problems. Within the closure operator approach, given a concrete interpretation $\langle C, \otimes \rangle$, an abstraction $A \in \mathcal{L}_C$ is complete when the equation $\rho_A \circ \otimes \circ \langle \rho_A, \rho_A \rangle = \rho_A \circ \otimes$ holds. Giacobazzi and Ranzato [10] stated the following *full completeness problem*, here specialized to binary semantic operations: Given an abstract domain $A \in \mathcal{L}_C$, does the following system with variable ρ admit a most abstract solution?

$$\begin{cases} \rho \sqsubseteq A \\ \rho \circ \otimes \circ \langle \rho, \rho \rangle = \rho \circ \otimes \end{cases} \quad (1)$$

Hence, a solution to the above full completeness problem is the necessarily unique (up to domain isomorphism) most abstract domain which includes A and induces a complete abstract interpretation. Following [10], such most abstract solution to System (1) is called the *least complete extension* of A w.r.t. \otimes . It is shown in [10] that if \otimes is continuous in both arguments, then least complete extensions of any A exist.

Full completeness problems clearly make sense only for concrete interpretations of type $\langle C, \otimes \rangle$. When generic concrete interpretations of type $\langle C, C_1, C_2, \otimes \rangle$ are considered, different abstractions are involved in a completeness equation, and therefore various completeness problems arise by fixing some of these abstractions. In the following, we introduce three such completeness problems, which turn out to be of particular interest. Such completeness problems still depend only on best correct approximations of the concrete operation, and therefore the corresponding completeness notions are again abstract domain properties.

Observation completeness problems. Let $\langle C, C_1, C_2, \otimes \rangle$ be a concrete interpretation. An observation domain is any abstraction of the range C of \otimes . Observation completeness problems arise when in a completeness equation an observation domain is fixed. Hence, let us consider a fixed observation domain $A \in \mathcal{L}_C$. The *observation completeness problem* for a pair $\langle A_1, A_2 \rangle \in \mathcal{L}_{C_1} \times \mathcal{L}_{C_2}$ admits solution when there exists the most abstract solution in $\mathcal{L}_{C_1} \times \mathcal{L}_{C_2}$ of the following system:

$$\begin{cases} \langle \eta, \mu \rangle \sqsubseteq \langle A_1, A_2 \rangle \\ \rho_A \circ \otimes \circ \langle \eta, \mu \rangle = \rho_A \circ \otimes \end{cases} \quad (2)$$

Let us remark that, by using adjunctions, the observation completeness equation $\rho_A \circ \otimes \circ \langle \eta, \mu \rangle = \rho_A \circ \otimes$ is equivalent to require that for all $x \in C_1$ and $y \in C_2$, $\alpha_{C_1, \eta}(x) \otimes^b \alpha_{C_2, \mu}(y) = \alpha_{C, A}(x \otimes y)$.

When in addition to the observation domain we also fix one (or more, if we would deal with n -ary operations) of the abstractions of the argument domains of \otimes , we obtain yet different completeness problems. In the *left observation completeness problem*, $A \in \mathcal{L}_C$ and $A_2 \in \mathcal{L}_{C_2}$ are fixed, and the solution to this problem for a given $A_1 \in \mathcal{L}_{C_1}$ exists when the following system with variable η admits a most abstract solution in \mathcal{L}_{C_1} :

$$\begin{cases} \eta \sqsubseteq A_1 \\ \rho_A \circ \otimes \circ \langle \eta, \rho_{A_2} \rangle = \rho_A \circ \otimes \circ \langle id, \rho_{A_2} \rangle \end{cases} \quad (3)$$

Of course, *right* observation completeness problems are analogously formulated. It turns out that a left observation completeness equation $\rho_A \circ \otimes \circ \langle \eta, \rho_{A_2} \rangle = \rho_A \circ \otimes \circ \langle id, \rho_{A_2} \rangle$ formulated in terms of GIs amounts to require that for any $x \in C_1$ and $y \in A_2$, $\alpha_{C_1, \eta}(x) \otimes^b y = \alpha_{C, A}(x \otimes \gamma_{A_2, C_2}(y))$. Hence, a left observation completeness equation $\rho_A \circ \otimes \circ \langle \eta, \rho_{A_2} \rangle = \rho_A \circ \otimes \circ \langle id, \rho_{A_2} \rangle$ states that completeness for the abstractions $A \in \mathcal{L}_C$, $\eta \in \mathcal{L}_{C_1}$ and $A_2 \in \mathcal{L}_{C_2}$ holds when the semantic operation \otimes acts over $C_1 \times A_2$. Examples of observation completeness problems will be considered and solved later in Section 5.

4 Quantales and Solutions to Completeness Problems

When concrete interpretations are quantales and typed quantales, solutions to the above completeness problems exist and can be characterized explicitly and elegantly in terms of linear implications.

Let $\langle C, C_1, C_2, \otimes \rangle$ be a typed quantale playing the rôle of concrete interpretation. In this setting, solutions to completeness problems will be characterized by exploiting two basic domain transformers $\overset{\wedge}{\rightarrow} : uco(C_1) \times uco(C) \rightarrow uco(C_2)$ and $\overset{\wedge}{\leftarrow} : uco(C) \times uco(C_2) \rightarrow uco(C_1)$, defined by lifting left and right linear implications \rightarrow and \leftarrow to abstract domains as follows: For any $A_1 \in \mathcal{L}_{C_1}$, $A_2 \in \mathcal{L}_{C_2}$, $A \in \mathcal{L}_C$:

$$\begin{aligned} A_1 \overset{\wedge}{\rightarrow} A &\stackrel{\text{def}}{=} \bigwedge (\{a_1 \rightarrow a \in C_2 \mid a_1 \in A_1, a \in A\}); \\ A \overset{\wedge}{\leftarrow} A_2 &\stackrel{\text{def}}{=} \bigwedge (\{a \leftarrow a_2 \in C_1 \mid a \in A, a_2 \in A_2\}). \end{aligned}$$

Hence, $A_1 \overset{\wedge}{\rightarrow} A$ is defined to be the most abstract domain in \mathcal{L}_{C_2} containing all the linear implications from A_1 to A . From the logic properties of linear implication recalled in Section 2, it is not too hard to derive the following useful distributivity laws for $\overset{\wedge}{\rightarrow}$ and $\overset{\wedge}{\leftarrow}$ over reduced product of abstract domains.

Proposition 1. *For all $\{B_i\}_{i \in I} \subseteq \mathcal{L}_C$,*

$$A_1 \overset{\wedge}{\rightarrow} (\prod_{i \in I} B_i) = \prod_{i \in I} (A_1 \overset{\wedge}{\rightarrow} B_i) \quad \text{and} \quad (\prod_{i \in I} B_i) \overset{\wedge}{\leftarrow} A_2 = \prod_{i \in I} (B_i \overset{\wedge}{\leftarrow} A_2).$$

Solutions to full completeness problems. Solutions to full completeness problems exist and are characterized in terms of linear implications among domain's objects as stated by the following result.

Theorem 2. $A \sqcap (C \xrightarrow{\wedge} A) \sqcap (A \xleftarrow{\wedge} C) \sqcap ((C \xrightarrow{\wedge} A) \xleftarrow{\wedge} C)$ is the most abstract solution of System [\(II\)](#).

Solutions to observation completeness problems. Let us first consider left observation completeness problems. A left observation completeness equation can be characterized as follows.

Theorem 3. $\rho_A \circ \otimes \circ \langle \eta, \rho_{A_2} \rangle = \rho_A \circ \otimes \circ \langle id, \rho_{A_2} \rangle \Leftrightarrow \eta \sqsubseteq A \xleftarrow{\wedge} A_2$.

Thus, as an immediate consequence, left observation completeness problems admit the following solutions.

Corollary 4. $A_1 \sqcap (A \xleftarrow{\wedge} A_2)$ is the most abstract solution of System [\(E\)](#).

Of course, dual results can be given for right observation completeness problems. In this case, the most abstract solution therefore is $A_2 \sqcap (A_1 \xrightarrow{\wedge} A)$.

The above results for left and right observation completeness turn out to be useful for solving observation completeness problems. In fact, an observation completeness equation is characterized as an independent combination of left and right observation completeness equations as follows.

Theorem 5. $\rho_A \circ \otimes \circ \langle \eta, \mu \rangle = \rho_A \circ \otimes \Leftrightarrow \langle \eta, \mu \rangle \sqsubseteq \langle A \xleftarrow{\wedge} C_2, C_1 \xrightarrow{\wedge} A \rangle$.

As a straight consequence, we get the following result.

Corollary 6. $\langle A_1 \sqcap (A \xleftarrow{\wedge} C_2), A_2 \sqcap (C_1 \xrightarrow{\wedge} A) \rangle$ is the most abstract solution of System [\(2\)](#).

4.1 The Case of Unital Commutative Quantales

When we deal with unital and commutative quantales — i.e. models of intuitionistic linear logic [\[16, 19\]](#) — the above solutions to full completeness problems given by Theorem [2](#) can be significantly simplified by exploiting the logical properties of linear implication. In a unital commutative quantale $\langle C, \otimes \rangle$, the following additional properties hold: For any $a, b, c \in C$:

- $a \rightarrow (b \rightarrow c) = b \rightarrow (a \rightarrow c)$ – $1 \rightarrow a = a$
- $c \leq (c \rightarrow a) \rightarrow a$ – $((c \rightarrow a) \rightarrow a) \rightarrow a = c \rightarrow a$

Therefore, from these properties it is not hard to check that for all $a \in C$, $\lambda c. (c \rightarrow a) \rightarrow a \in uco(C)$. This turns out to be the key observation in order to give a more compact form to solutions of full completeness problems on unital commutative quantales. Moreover, the objects of such solutions enjoy a clean logical characterization in terms of linear implications as specified by the third point of the next result.

Theorem 7. Let $\langle C, \otimes \rangle$ be a unital commutative quantale.

1. $C \xrightarrow{\wedge} A$ is the most abstract solution of System (II);
2. If $B \in \mathcal{L}_C$ is such that $B \sqsubseteq C \xrightarrow{\wedge} A$, then $C \xrightarrow{\wedge} A = B \xrightarrow{\wedge} A$;
3. For all $c \in C$, $c \in C \xrightarrow{\wedge} A \Leftrightarrow c = \bigwedge_{a \in A} (c \rightarrow a) \rightarrow a$.

5 An Application in Data Structure Completeness

In this section, we consider some examples of completeness problems in abstract interpretation of list data structures. For the sake of practicality, we consider lists of natural numbers, even if our discussion holds more in general for lists of objects of arbitrary type.

Consider the structure $\langle \wp(list(\mathbb{N})), \wp(\mathbb{N}), \wp(list(\mathbb{N})), :: \rangle$, where $list(\mathbb{N})$ is the set of all finite lists of natural numbers, $\wp(list(\mathbb{N}))$ and $\wp(\mathbb{N})$ are complete lattices w.r.t. set-inclusion, and $:: : \wp(\mathbb{N}) \times \wp(list(\mathbb{N})) \rightarrow \wp(list(\mathbb{N}))$ is a “collecting” version of concatenation defined as follows:

$$N :: L \stackrel{\text{def}}{=} \{[n|l] \mid n \in N, l \in L\},$$

where $\emptyset :: L = N :: \emptyset = \emptyset$. It is clear that this structure is a typed quantale.

We say that a list is *irredundant* if it does not contain two occurrences of the same object. An abstract domain $\rho \in uco(\wp(list(\mathbb{N})))$ for detecting irredundant lists can be defined by $\rho \stackrel{\text{def}}{=} \{list(\mathbb{N}), Irr\}$, where $Irr \subseteq list(\mathbb{N})$ is the set of irredundant lists over \mathbb{N} . We consider ρ as an observation domain and we look for the most abstract solution $\langle X, Y \rangle \in uco(\wp(\mathbb{N})) \times uco(\wp(list(\mathbb{N})))$ to the following observation completeness problem:

$$\rho \circ :: \circ \langle \rho_X, \rho_Y \rangle = \rho \circ ::$$

Here, we dropped the first constraint of the generic System (2), since it amounts to the trivial constraint $\langle \rho_X, \rho_Y \rangle \sqsubseteq \langle \{\mathbb{N}\}, \{list(\mathbb{N})\} \rangle$ which is always satisfied. By Corollary 6 the most abstract solution of this observation completeness equation exists. This will be the most abstract pair of domains $\langle X, Y \rangle$ for which abstract concatenation in X and Y results to be complete when observing irredundancy as represented by ρ . By Corollary 6, the solution $\langle X, Y \rangle$ is as follows.

$$\begin{aligned} X &= \{\mathbb{N}\} \sqcap (\rho \xleftarrow{\wedge} \wp(list(\mathbb{N}))) \\ &= \rho \xleftarrow{\wedge} \wp(list(\mathbb{N})) \\ &= \bigwedge (\{L \leftarrow M \mid L \in \rho, M \in \wp(list(\mathbb{N}))\}) \\ &\quad (\text{since, for all } M, list(\mathbb{N}) \leftarrow M = list(\mathbb{N})) \\ &= \bigwedge (\{Irr \leftarrow M \mid M \in \wp(list(\mathbb{N}))\}) \\ &\quad (\text{since, for all } n \in \mathbb{N}, Irr \leftarrow \{[n]\} = \mathbb{N} \setminus \{n\}) \\ &= \wp(\mathbb{N}) \end{aligned}$$

$$\begin{aligned}
Y &= \{ \text{list}(\mathbb{N}) \} \sqcap (\wp(\mathbb{N}) \xrightarrow{\wedge} \rho) \\
&= \wp(\mathbb{N}) \xrightarrow{\wedge} \rho \\
&= \bigwedge (\{ N \rightarrow L \mid N \in \wp(\mathbb{N}), L \in \rho \}) \\
&\quad (\text{since, for all } N, N \rightarrow \text{list}(\mathbb{N}) = \text{list}(\mathbb{N})) \\
&= \bigwedge (\{ N \rightarrow \text{Irr} \mid N \in \wp(\mathbb{N}) \}) \\
&\quad (\text{since, by (v) in Section 2, } \bigcap_i (N_i \rightarrow \text{Irr}) = (\bigcup_i N_i \rightarrow \text{Irr})) \\
&= \{ N \rightarrow \text{Irr} \mid N \in \wp(\mathbb{N}) \} \\
&= \bigcup_{N \subseteq \mathbb{N}} \{ L \in \wp(\text{list}(\mathbb{N})) \mid l \in L \Leftrightarrow (l \in \text{Irr} \text{ and } \forall n \in N. n \text{ is not in } l) \}
\end{aligned}$$

Thus, in order to be complete for concatenation when observing irredundancy, X must coincide with the concrete domain $\wp(\mathbb{N})$, while it suffices that Y contains all the sets of irredundant lists which do not contain some set of numbers. Note that Y coincides with the set of all the sets of irredundant lists closed by permutation of their objects, and this is a strict abstraction of the concrete domain $\wp(\text{list}(\mathbb{N}))$.

Let us now consider the standard abstract domain $\eta \in \text{uco}(\wp(\mathbb{N}))$ for parity analysis given by $\eta \stackrel{\text{def}}{=} \{ \mathbb{N}, \text{even}, \text{odd}, \emptyset \}$. We consider the following left and right observation completeness problems: We look respectively for the the most abstract domains $X \in \text{uco}(\wp(\mathbb{N}))$ and $Y \in \text{uco}(\wp(\text{list}(\mathbb{N})))$ such that:

$$(i) \quad \rho \circ :: \circ \langle \rho_X, \rho \rangle = \rho \circ :: \langle id, \rho \rangle \quad (ii) \quad \rho \circ :: \circ \langle \eta, \rho_Y \rangle = \rho \circ :: \langle \eta, id \rangle$$

Here again, there are no upper bound constraints for X and Y . By Corollary 4 (and its dual), we get the following solutions:

$$\begin{aligned}
X &= \{ \mathbb{N} \} \sqcap (\rho \xleftarrow{\wedge} \rho) \\
&= \rho \xleftarrow{\wedge} \rho \\
&= \bigwedge (\{ \text{list}(\mathbb{N}) \leftarrow \text{list}(\mathbb{N}), \text{list}(\mathbb{N}) \leftarrow \text{Irr}, \text{Irr} \leftarrow \text{list}(\mathbb{N}), \text{Irr} \leftarrow \text{Irr} \}) \\
&= \{ \mathbb{N}, \emptyset \}, \\
Y &= \{ \text{list}(\mathbb{N}) \} \sqcap (\eta \xrightarrow{\wedge} \rho) \\
&= \eta \xrightarrow{\wedge} \rho \\
&= \bigwedge (\{ \mathbb{N} \rightarrow \text{Irr}, \text{even} \rightarrow \text{Irr}, \text{odd} \rightarrow \text{Irr}, \emptyset \rightarrow \text{Irr} \}) \\
&= \{ \{ \} \}, \text{Irr}_{\text{even}}, \text{Irr}_{\text{odd}}, \text{list}(\mathbb{N}) \},
\end{aligned}$$

where $\text{Irr}_{\text{even}} \stackrel{\text{def}}{=} \{ l \in \text{list}(\mathbb{N}) \mid l \in \text{Irr} \text{ and } l \text{ does not contain even numbers} \}$ and $\text{Irr}_{\text{odd}} \stackrel{\text{def}}{=} \{ l \in \text{list}(\mathbb{N}) \mid l \in \text{Irr} \text{ and } l \text{ does not contain odd numbers} \}$. Thus, for problem (i), in order to get completeness, it is enough to check whether a given set of numbers is empty or not, while, for problem (ii), we only need to consider sets of irredundant lists which do not contain either even or odd numbers.

6 Complete Semantics for Logic Program Analysis

In this section, we determine the least complete extension of any logic program property w.r.t. a bottom-up semantics characterizing computed answer substitutions, which turns out to be equivalent to the well-known s-semantics [8]. This

complete semantics can be thought of as a logic program semantics which is “optimal” (i.e. neither too concrete nor too abstract) for characterizing a given property of interest. The problem of determining optimal semantics for program analysis was raised in [9]. We show that such optimal semantics can be obtained by solving a full completeness problem relatively to the operation of unification on sets of substitutions.

Completeness for an abstract domain, in general, depends both on the considered concrete domains and on the semantic operation defined on them. However, the following important result shows that, under certain conditions, completeness is instead independent from the choice of the concrete semantics.

Theorem 8. *Let $\alpha \in \mathcal{L}_C$ and $f, g : C \rightarrow C$ such that $f \sqsubseteq g \sqsubseteq \alpha \circ f$. Then, α is complete for f iff α is complete for g .*

We will exploit this result for computing our least complete extension of abstract domains w.r.t. s-semantics. The idea is that of considering a “simplified” and more concrete version, denoted by T_P , of the immediate consequences operator T_P^s of s-semantics, which does not take into account variable renaming. This will simplify a lot the technical development of this section. Hence, s-semantics results to be an abstract interpretation of our T_P semantics: If r denotes the closure under variable renaming of sets of substitutions, then we have that $T_P^s = r \circ T_P$. Then, leaving out the details, if α denotes the least complete extension, relatively to T_P , of any domain A abstracting computed answer substitutions, since $T_P \sqsubseteq r \circ T_P = T_P^s \sqsubseteq \alpha \circ T_P$ holds, we can apply Theorem 8, from which we get that α not only is complete for T_P^s , but actually α turns out to be the least complete extension of A relatively to T_P^s .

6.1 Notation

Let \mathcal{V} be an infinite, recursively enumerable (r.e.) set of variables, Σ be a set of function symbols and Π be a set of predicate symbols, defining a r.e. first-order language \mathcal{L} . *Term* denotes the set of terms of \mathcal{L} . If s is any syntactic object and σ and θ are substitutions, then $vars(s)$ denotes the set of variables occurring in s , $dom(\sigma) \stackrel{\text{def}}{=} \{v \in \mathcal{V} \mid \sigma(v) \neq v\}$, $rng(\sigma) \stackrel{\text{def}}{=} \bigcup \{vars(\sigma(x)) \mid x \in dom(\sigma)\}$, $s\sigma$ denotes the application of σ to s , and $\sigma \circ \theta$ denotes the standard composition of θ and σ (i.e., $\sigma \circ \theta = \lambda x.(\theta(x))\sigma$). The set of idempotent substitutions modulo renaming (i.e., $\theta \sim \sigma$ iff there exists β and δ such that $\theta = \sigma \circ \beta$ and $\sigma = \theta \circ \delta$) on \mathcal{L} is denoted by *Sub*. Objects in *Sub* are partially ordered by instantiation, denoted by \preceq . By adding to *Sub* an extra object τ as least element, one gets a complete lattice $\langle Sub^\tau, \preceq, \vee, \wedge, \epsilon, \tau \rangle$, where \vee is least general anti-instance, \wedge is standard unification and ϵ is the empty substitution (see [14] for more details). The set of most general atoms is given by $GAtom \stackrel{\text{def}}{=} \{p(\bar{X}) \mid p \in \Pi\}$, where \bar{X} is a tuple of distinct variables. We consider logic programs in normalized form, that is, a generic Horn clause is $p(\bar{X}) : -c, q_1(\bar{X}_1), \dots, q_n(\bar{X}_n)$, where all the tuples of variables are distinct and $c \in Sub$ is an idempotent substitution binding variables to terms.

6.2 T_P -Completeness

Our basic semantic structure is the unital commutative quantale $\langle \wp(\text{Sub}), \otimes \rangle$, where $\langle \wp(\text{Sub}), \subseteq \rangle$ is a complete lattice, $\otimes : \wp(\text{Sub}) \times \wp(\text{Sub}) \rightarrow \wp(\text{Sub})$ is the obvious lifting of unification \wedge to sets of substitutions, i.e. it is defined by: $X \otimes Y \stackrel{\text{def}}{=} \{x \wedge y \mid x \in X, y \in Y\}$, and $\{\epsilon\} \in \wp(\text{Sub})$ is the unit of \otimes . It is immediate to check that $\langle \wp(\text{Sub}), \otimes \rangle$ actually is a unital commutative quantale [16, Example 10, p. 18]. In the following, we will slightly abuse notation by applying the operation \otimes also to substitutions.

As mentioned above, we consider a bottom-up semantics based on an immediate consequences operator T_P which is more concrete than the standard operator T_P^s of s-semantics. In fact, T_P is defined using only the operations of unification \otimes and union of sets of idempotent substitutions. The s-semantics operator T_P^s can be recovered from T_P by a simple step of abstract interpretation considering the closure of sets of substitutions over variables renaming.

We consider a concrete semantic domain $CInt$ of functions — as usual, called interpretations — which map most general atoms to sets of substitutions: $CInt \stackrel{\text{def}}{=} GAtom \rightarrow \wp(\text{Sub})$, which ordered pointwise is trivially a complete lattice. Often, we will find convenient to denote an interpretation $I \in CInt$ by the set $\{\langle p(\bar{X}), I(p(\bar{X})) \rangle \mid p \in \Pi\}$. Then, for any program P , the immediate consequences operator $T_P : CInt \rightarrow CInt$ is defined as follows: For any $I \in CInt$,

$$T_P(I)(p(\bar{Y})) = \bigcup_{C \lll P} (c \otimes (\bigotimes_{i=1..n} I(q_i(\bar{X}_i))) \otimes \{\bar{Y} = \bar{X}\}),$$

where $C = p(\bar{X}) : -c, q_1(\bar{X}_1), \dots, q_n(\bar{X}_n)$. Here, $C \lll P$ denotes that the clause C of P is renamed apart with fresh variables, and $I(q_i(\bar{X}_i))$ is intended modulo renaming.

If $\rho \in uco(\wp(\text{Sub}))$ is any abstraction on sets of substitutions, a corresponding abstraction on interpretations $\{\rho\} \in uco(CInt)$ which acts accordingly to ρ can be defined as follows: For any $I \in CInt$, $\{\rho\}(I) \stackrel{\text{def}}{=} \{\langle p(\bar{X}), \rho(c) \rangle \mid \langle p(\bar{X}), c \rangle \in I\}$. Note that $\{\cdot\}$ is monotone, i.e., for all $\rho, \eta \in uco(\wp(\text{Sub}))$, if $\rho \sqsubseteq \eta$ then $\{\rho\} \sqsubseteq \{\eta\}$.

Given a basic abstract domain of properties of interest $\pi \in uco(\langle \wp(\text{Sub}), \subseteq \rangle)$, our goal is therefore to find the most abstract domain which contains π (more precisely, $\{\pi\}$) and is complete for any T_P operator. Our strategy consists in characterizing the least complete extension of π for the basic operations involved in the definition of T_P , and then to show that, under reasonable hypotheses, this domain turns out to be the right one. Since every abstract domain is trivially always complete for lub's, it turns out that union of sets of substitutions is troubleless, and therefore it is enough to concentrate on unification \otimes . Indeed, the following result shows that completeness for \otimes implies completeness for any T_P .

Theorem 9. *Let $\rho \in uco(\wp(\text{Sub}))$. If ρ is complete for \otimes then $\{\rho\}$ is complete for any T_P .*

By Theorem 7 given $\pi \in uco(\wp(Sub))$, the least complete extension of π w.r.t. the quantale operation \otimes exists and is the domain $\wp(Sub) \xrightarrow{\wedge} \pi$ of linear implications from $\wp(Sub)$ to π . This complete domain $\wp(Sub) \xrightarrow{\wedge} \pi$ results to be the right one whenever the abstract domain π satisfies the following weak decidability property.

Definition 10. $\pi \in uco(\wp(Sub))$ is decidable if any $S \in \pi$ is a r.e. set. \square

It should be clear that decidability is a reasonable requirement for most abstract domains used in program analysis: In fact, for such a decidable abstraction (of sets) of substitutions, an effective procedure for checking whether a substitution belongs to (is approximated by) an abstract object is available. As announced, the following key result shows that least complete extensions of decidable abstract domains w.r.t. unification \otimes actually are least complete extensions for T_P operators as well.

Theorem 11. Let $\pi \in uco(\wp(Sub))$ be decidable. Then, $\wp(Sub) \xrightarrow{\wedge} \pi$ is the least complete extension of \wp for any T_P .

As we discussed just after Theorem 8 it turns out that, for any decidable $\pi \in uco(\wp(Sub))$, $\wp(Sub) \xrightarrow{\wedge} \pi$ actually is the least complete extension of \wp for any immediate consequences operator T_P^s of s-semantics. In fact, if $r \in uco(\wp(Sub))$ is the closure under renaming of sets of substitutions, namely variables in the range of substitutions are renamed in each possible way, then $T_P^s = r \circ T_P$; moreover, $\wp(Sub) \xrightarrow{\wedge} \pi$ clearly induces a semantic transformer less precise than T_P^s , i.e. $T_P^s \sqsubseteq \wp(Sub) \xrightarrow{\wedge} \pi \circ T_P$. Hence, by Theorems 8 and 11, we get the following desired consequence.

Corollary 12. Let $\pi \in uco(\wp(Sub))$ be decidable, such that $T_P^s \sqsubseteq \pi \circ T_P$ for any P . Then, $\wp(Sub) \xrightarrow{\wedge} \pi$ is the least complete extension of \wp for any T_P^s .

6.3 Complete Semantics for Groundness Analysis

Groundness analysis is arguably one of the most important analysis for logic-based programming languages. Groundness analysis aims to statically detect whether variables will be bound to ground terms in successful derivations. By instantiating the results above, we are able to characterize the least complete extension of the basic abstract domain representing plain groundness information w.r.t. any immediate consequences operator of s-semantics. The resulting semantics can be therefore interpreted as the “optimal” semantics for groundness analysis.

If $V \subseteq \mathcal{V}$ is a finite set of variables of interest, the simplest abstract domain for representing plain groundness information of variables in V is $\mathcal{G}_V \stackrel{\text{def}}{=} \langle \wp(V), \supseteq \rangle$, as first put forward by Jones and Søndergaard [12]. The intuition is that each $W \in \mathcal{G}_V$ represents the set of substitutions which ground every variable in W . \mathcal{G}_V is related to the concrete domain $\wp(Sub)$ by the following concretization map: For each $W \in \mathcal{G}$, $\gamma_{\mathcal{G}_V}(W) \stackrel{\text{def}}{=} \{\theta \in Sub \mid \forall v \in W. vars(\theta(v)) = \emptyset\}$. As usual, we

shall abuse notation by denoting with $\mathcal{G}_V \in uco(\wp(Sub))$ the corresponding isomorphic image $\gamma_{\mathcal{G}_V}(\mathcal{G}_V)$ in $\wp(Sub)$.

A variable independent abstract domain $\mathcal{G} \in uco(CInt)$ for representing plain groundness information can be therefore defined as follows:

$$\mathcal{G}(I) \stackrel{\text{def}}{=} \{ \langle p(\bar{X}), \mathcal{G}_{\bar{X}}(c) \rangle \mid \langle p(\bar{X}), c \rangle \in I \}.$$

It is easy to check that \mathcal{G} actually is a uco on the concrete semantic domain $CInt$. Furthermore, \mathcal{G} is clearly decidable and $T_P^s \subseteq \mathcal{G} \circ T_P$. Thus, as an easy consequence of Corollary 12 we can prove the following result, where $V \subset_f \mathcal{V}$ means that V is a finite subset of \mathcal{V} .

Theorem 13. $\sqcap_{V \subset_f \mathcal{V}} \{ \wp(Sub) \xrightarrow{\lambda} \mathcal{G}_V \}$ is the least complete extension of \mathcal{G} for any T_P^s .

References

1. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: theory and applications. *J. Logic Program.*, 19-20:149–197, 1994. 218
2. P. Cousot. Completeness in abstract interpretation (Invited Talk). In *Proc. 1995 Joint Italian-Spanish Conference on Declarative Programming*, pages 37–38, 1995. 216
3. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation (Invited Paper). In *Proc. of the 13th Int. Symp. on Math. Found. of Programming Semantics (MFPS'97)*, vol. 6 of Electronic Notes in Theor. Comput. Sci., 1997. 216
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM POPL*, pages 238–252, 1977. 216, 218
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM POPL*, pages 269–282, 1979. 215, 216, 218, 218, 220
6. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proc. 19th ACM POPL*, pages 83–94, 1992. 216
7. P. Cousot and R. Cousot. Abstract interpretation of algebraic polynomial systems. In *Proc. 6th Int. Conf. on Algebraic Methodology and Software Technology (AMAST'97)*, LNCS 1349, pages 138–154, 1997. 216
8. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theor. Comput. Sci.*, 69(3):289–318, 1989. 218, 224
9. R. Giacobazzi. “Optimal” collecting semantics for analysis in a hierarchy of logic program semantics. In *Proc. 13th Int. Symp. on Theor. Aspects of Comput. Sci. (STACS'96)*, LNCS 1046, pages 503–514, 1996. 216, 225
10. R. Giacobazzi and F. Ranzato. Completeness in abstract interpretation: a domain perspective. In *Proc. 6th Int. Conf. on Algebraic Methodology and Software Technology (AMAST'97)*, LNCS 1349, pages 231–245, 1997. 216, 216, 216, 216, 220, 220, 220, 220
11. R. Giacobazzi, F. Ranzato and F. Scozzari. Complete abstract interpretations made constructive. In *Proc. 23rd Int. Symp. on Math. Found. of Comp. Sci. (MFCS'98)*, LNCS, 1998. 216, 217

12. N.D. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood Ltd, 1987. [227](#)
13. A. Mycroft. Completeness and predicate-based abstract interpretation. In *Proc. ACM Conf. on Partial Evaluation and Program Manipulation (PEPM'93)*, pages 179–185, 1993. [216](#), [216](#), [216](#)
14. C. Palamidessi. Algebraic properties of idempotent substitutions. In *Proc. 17th Int. Colloq. on Automata, Languages and Programming (ICALP'90)*, LNCS 443, pages 386–399, 1990. [225](#)
15. U.S. Reddy and S.N. Kamin. On the power of abstract interpretation. *Computer Languages*, 19(2):79–89, 1993. [216](#)
16. K.I Rosenthal. *Quantales and their Applications*. Longman Scientific & Technical, 1990. [217](#), [219](#), [219](#), [222](#), [226](#)
17. R.C. Sekar, P. Mishra, and I.V. Ramakrishnan. On the power and limitation of strictness analysis. *J. ACM*, 44(3):505–525, 1997. [216](#)
18. B. Steffen. Optimal data flow analysis via observational equivalence. In *Proc. 14th Int. Symp. on Math. Found. of Comp. Sci. (MFCS'89)*, LNCS 379, pages 492–502, 1989. [216](#)
19. D. Yetter. Quantales and (noncommutative) linear logic. *J. Symbolic Logic*, 55(1):41–64, 1990. [217](#), [219](#), [222](#)

On the Power of Homeomorphic Embedding for Online Termination

Michael Leuschel*

Department of Computer Science, K.U. Leuven, Belgium
Department of Electronics and Computer Science, University of Southampton, UK
DIKU, University of Copenhagen, Denmark
`mal@ecs.soton.ac.uk`

Abstract. Recently well-quasi orders in general, and homeomorphic embedding in particular, have gained popularity to ensure the termination of program analysis, specialisation and transformation techniques. In this paper we investigate and clarify for the first time, both intuitively and formally, the advantages of such an approach over one using well-founded orders. Notably we show that the homeomorphic embedding relation is strictly more powerful than a large class of involved well-founded approaches.

1 Introduction

The problem of ensuring termination arises in many areas of computer science and a lot of work has been devoted to proving termination of term rewriting systems (e.g. [7,8,9,37] and references therein) or of logic programs (e.g. [6,38] and references therein). It is also an important issue within all areas of program analysis, specialisation and transformation: one usually strives for methods which are guaranteed to terminate. One can basically distinguish between two kinds of techniques for ensuring termination:

- *static* techniques, which prove or ensure termination of a program or process *beforehand* (i.e. *off-line*) without any kind of execution, and
- *online* (or *dynamic*) techniques, which ensure termination of a process *during* its execution. (The process itself can of course be, e.g., performing a static analysis.)

Static approaches have less information at their disposal but do not require runtime intervention (which might be impossible). Which of the two approaches is taken depends entirely on the application area. For instance, static termination analysis of logic programs [6,38] falls within the former context, while termination of program specialisation, transformation or analysis is often ensured in an online manner.

This paper is primarily aimed at studying and improving online termination techniques. Let us examine the case of partial deduction [29,10,23] — an automatic technique for specialising logic programs. Henceforth we suppose some familiarity with basic notions in logic programming [3,28].

* Part of the work was done while the author was Post-doctoral Fellow of the Fund for Scientific Research - Flanders Belgium (FWO) and visiting DIKU, University of Copenhagen.

Partial deduction based upon the Lloyd and Shepherdson framework [29] generates (possibly incomplete) SLDNF-trees for a set \mathcal{A} of atoms. The specialised program is extracted from these trees by producing one clause for every non-failing branch. The resolution steps within the SLDNF-trees — often referred to as *unfolding* steps — are those that have been performed beforehand, justifying the hope that the specialised program is more efficient.

Now, to ensure termination of partial deduction two issues arise [10, 34]. One is called the *local termination* problem, corresponding to the fact that each generated SLDNF-tree should be finite. The other is called the *global termination* problem, meaning that the set \mathcal{A} should contain only a finite number of atoms. A similar classification can be done for most other program specialisation techniques (cf. e.g. [26]).

Below we mainly use local termination to illustrate our concepts. (As shown in [34] the atoms in \mathcal{A} can be structured into a global tree and methods similar to the one for local termination can be used to ensure global termination.)

However, the discussions and contributions of the present paper are also (immediately) applicable in the context of analysis, specialisation and transformation techniques in general, especially when applied to computational paradigms, such as logic programming, constrained logic programming, conditional term rewriting, functional programming and functional & logic programming. For instance, abstract interpretation techniques usually analyse a set of abstract calls to which new call patterns are continuously added. One thus faces a problem very similar to global termination of partial deduction.

Depth Bounds. One, albeit ad-hoc, way to solve the local termination problem is to simply impose an arbitrary *depth bound*. Such a depth bound is of course not motivated by any property, structural or otherwise, of the program or goal under consideration. In the context of local termination, the depth bound will therefore typically lead either to too little or too much unfolding.

Determinacy. Another approach, often used in partial evaluation of functional programs [17] is to (only) expand a tree while it is *determinate* (i.e. it only has one non-failing branch). However, this approach can be very restrictive and in itself does not guarantee termination, as there can be infinitely failing determinate computations at specialisation time.

Well-Founded Orders. Luckily, more refined approaches to ensure local termination exist. The first non-ad-hoc methods [5, 33, 32, 31] in logic and [40, 47] functional programming were based on *well-founded orders*, inspired by their usefulness in the context of static termination analysis. These techniques ensure termination, while at the same time allowing unfolding related to the structural aspect of the program and goal to be specialised, e.g., permitting the consumption of static input within the atoms of \mathcal{A} .

Definition 1. (wfo) A (strict) partial order $>_S$ on a set S is an *anti-reflexive*, *anti-symmetric* and *transitive* binary relation on $S \times S$. A sequence of elements s_1, s_2, \dots in S is called *admissible wrt $>_S$* iff $s_i > s_{i+1}$, for all $i \geq 1$. We call $>_S$ a *well-founded order (wfo)* iff there is no infinite admissible sequence wrt $>_S$.

To ensure local termination, one has to find a sensible well-founded order on atoms and then only allow SLDNF-trees in which the sequence of selected atoms is admissible wrt the well-founded order. If an atom that we want to select is not strictly smaller than its ancestors, we either have to select another atom or stop unfolding altogether.

Example 1. Let P be the *reverse* program using an accumulator:

$$\begin{aligned} \text{rev}([], \text{Acc}, \text{Acc}) &\leftarrow \\ \text{rev}([H|T], \text{Acc}, \text{Res}) &\leftarrow \text{rev}(T, [H|\text{Acc}], \text{Res}) \end{aligned}$$

A simple well-founded order on atoms of the form $\text{rev}(t_1, t_2, t_3)$ might be based on comparing the term size (i.e., the number of function and constant symbols) of the first argument. We then define the wfo on atoms by:

$$\text{rev}(t_1, t_2, t_3) > \text{rev}(s_1, s_2, s_3) \text{ iff } \text{term_size}(t_2) > \text{term_size}(s_2).$$

Based on that wfo, the goal $\leftarrow \text{rev}([a, b|T], [], R)$ can be unfolded into the goal $\leftarrow \text{rev}([b|T], [a], R)$ and further into $\leftarrow \text{rev}(T, [b, a], R)$ because the term size of the first argument strictly decreases at each step (even though the overall term size does not decrease). However, $\leftarrow \text{rev}(T, [b, a], R)$ cannot be further unfolded into $\leftarrow \text{rev}(T', [H', b, a], R)$ because there is no such strict decrease.

Much more elaborate techniques, which e.g. split the expressions into classes, use lexicographical ordering on subsequences of the arguments and even continuously refine the orders during the unfolding process, exist and we refer the reader to [5, 33, 32, 31] for precise details. These works also present some further refinements on *how to apply* wfo's, especially in the context of partial deduction. For instance, instead of requiring a decrease wrt every ancestor, one can only request a decrease wrt the *covering ancestors*, i.e. one only compares with the ancestor atoms from which the current atom descends (via resolution). Other refinements consist in allowing the wfo's not only to depend upon the selected atom but on the *context* as well [32] or to ignore calls to *non-recursive* predicates. [32] also discusses a way to relax the condition of a “strict decrease” when refining a wfo. (Most of these refinements can also be applied to other approaches, notably the one we will present in the next section.)

However, it has been felt by several researchers that well-founded orders are sometimes too rigid or (conceptually) too complex in an online setting. Recently, well-quasi orders have therefore gained popularity to ensure online termination of program manipulation techniques [4, 41, 42, 25, 26, 11, 18, 120, 46]. Unfortunately, this move to well-quasi orders has never been formally justified nor has the relation to well-founded approaches been investigated. We strive to do so in this paper and will actually prove that a rather simple well-quasi approach is strictly more powerful than a large class of involved well-founded approaches.

2 Well-Quasi Orders and Homeomorphic Embedding

Formally, well-quasi orders can be defined as follows.

Definition 2. (quasi order) A quasi order \geq_S on a set S is a reflexive and transitive binary relation on $S \times S$.

Henceforth, we will use symbols like $<$, $>$ (possibly annotated by some subscript) to refer to strict partial orders and \leq , \geq to refer to quasi orders. We will use either “directionality” as is convenient in the context. We also define an *expression* to be either a *term* (built-up from variables and function symbols of arity ≥ 0) or an *atom* (a predicate symbol applied to a, possibly empty, sequence of terms), and then treat predicate symbols as functors, but suppose that no confusion between function and predicate symbols can arise (i.e. predicate and function symbols are distinct).

Definition 3. (wbr, wqo) Let \leq_S be a binary relation on $S \times S$. A sequence of elements s_1, s_2, \dots in S is called *admissible wrt* \leq_S iff there are no $i < j$ such that $s_i \leq_S s_j$. We say that \leq_S is a *well-binary relation (wbr)* on S iff there are no infinite admissible sequences wrt \leq_S . If \leq_S is a quasi order on S then we also say that \leq_S is a *well-quasi order (wqo)* on S .

Observe that, in contrast to wfo’s, non-comparable elements are allowed within admissible sequences. An admissible sequence is sometimes called *bad* while a non-admissible one is called *good*. A well-binary relation is then such that all infinite sequences are good. There are several other equivalent definitions of well-binary relations and well-quasi orders. Higman [14] used an alternate definition of well-quasi orders in terms of the “finite basis property” (or “finite generating set” in [19]). A different (but also equivalent) definition of a wqo is: A quasi-order \leq_V is a wqo iff for all quasi-orders \preceq_V which contain \leq_V (i.e. $v \leq_V v' \Rightarrow v \preceq_V v'$) the corresponding strict partial order \prec_V is a wfo. This property has been exploited in the context of *static* termination analysis to dynamically construct well-founded orders from well-quasi ones and led to the initial use of wqo’s in the offline setting [7,8]. The use of well-quasi orders in an *online* setting has only emerged recently (it is mentioned, e.g., in [4] but also [41]) and has never been compared to well-founded approaches. There has been some comparison between wfo’s and wqo’s in the offline setting, e.g., in [37] it is argued that (for “simply terminating” rewrite systems) approaches based upon quasi-orders are less interesting than ones based upon a partial orders. In this paper we will show that the situation is somewhat reversed in an online setting. Furthermore, in the online setting, transitivity of a wqo is not really interesting and one can therefore drop this requirement, leading to the use of wbr’s. [24] contains some useful wbr’s which are not wqo’s.

An interesting wqo is the homeomorphic embedding relation \sqsubseteq , which derives from results by Higman [14] and Kruskal [19]. It has been used in the context of term rewriting systems in [7,8], and adapted for use in supercompilation [45] in [42]. Its usefulness as a stop criterion for partial evaluation is also discussed and

advocated in [30]. Some complexity results can be found in [44] and [13] (also summarised in [30]).

The following is the definition from [42], which adapts the pure homeomorphic embedding from [8] by adding a rudimentary treatment of variables.

Definition 4. (\sqsubseteq) *The (pure) homeomorphic embedding relation \sqsubseteq on expressions is defined inductively as follows (i.e. \sqsubseteq is the least relation satisfying the rules):*

1. $X \sqsubseteq Y$ for all variables X, Y
2. $s \sqsubseteq f(t_1, \dots, t_n)$ if $s \sqsubseteq t_i$ for some i
3. $f(s_1, \dots, s_n) \sqsubseteq f(t_1, \dots, t_n)$ if $\forall i \in \{1, \dots, n\} : s_i \sqsubseteq t_i$.

The second rule is sometimes called the *diving* rule, and the third rule is sometimes called the *coupling* rule. When $s \sqsubseteq t$ we also say that s is *embedded in* t or t is *embedding* s . By $s \triangleleft t$ we denote that $s \sqsubseteq t$ and $t \not\sqsubseteq s$.

Example 2. The intuition behind the above definition is that $A \sqsubseteq B$ iff A can be obtained from B by “striking out” certain parts, or said another way, the structure of A reappears within B . For instance we have $p(a) \sqsubseteq p(f(a))$ and indeed $p(a)$ can be obtained from $p(f(a))$ by “striking out” the f . Observe that the “striking out” corresponds to the application of the diving rule 2 and that we even have $p(a) \triangleleft p(f(a))$. We also have, e.g., that:

$X \sqsubseteq X$, $p(X) \triangleleft p(f(Y))$, $p(X, X) \sqsubseteq p(X, Y)$ and $p(X, Y) \sqsubseteq p(X, X)$.

Proposition 1. *The relation \sqsubseteq is a wqo on the set of expressions over a finite alphabet.*

For a complete proof, reusing Higman’s and Kruskal’s results [14,19] in a very straightforward manner, see, e.g., [23]. Extensions to infinite alphabets and improved treatment of variables can be found in [24].

To ensure, e.g., local termination of partial deduction, we have to ensure that the constructed SLDNF-trees are such that the selected atoms do *not embed* any of their ancestors (when using a well-founded order as in Example 1 we had to require a *strict decrease* at every step). If an atom that we want to select embeds one of its ancestors, we either have to select another atom or stop unfolding altogether. For example, based on \sqsubseteq , the goal $\leftarrow \text{rev}([a, b|T], [], R)$ of Example 1 can be unfolded into $\leftarrow \text{rev}([b|T], [a], R)$ and further into $\leftarrow \text{rev}(T, [b, a], R)$ as $\text{rev}([a, b|T], [], R) \not\sqsubseteq \text{rev}([b|T], [a], R)$, $\text{rev}([a, b|T], [], R) \not\sqsubseteq \text{rev}(T, [b, a], R)$ and $\text{rev}([b|T], [a], R) \not\sqsubseteq \text{rev}(T, [b, a], R)$. However, $\leftarrow \text{rev}(T, [b, a], R)$ cannot be further unfolded into $\leftarrow \text{rev}(T', [H', b, a], R)$ as $\text{rev}(T, [b, a], R) \sqsubseteq \text{rev}(T', [H', b, a], R)$. Observe that, in contrast to Example 1, we did not have to choose how to measure which arguments. We further elaborate on the inherent flexibility of \sqsubseteq in the next section.

The homeomorphic embedding relation is also useful for handling structures other than expressions. It has, e.g., been successfully applied in [25,23,26] to detect (potentially) non-terminating sequences of characteristic trees. Also, \sqsubseteq seems to have the desired property that very often only “real” loops are detected and that they are detected at the earliest possible moment (see [30]).

3 Comparing wbr's and wfo's

3.1 General Comparison

It follows from Definitions 1 and 3 that if \leq_V is a wqo then $<_V$ (defined by $v_1 <_V v_2$ iff $v_1 \leq_V v_2 \wedge v_1 \not\leq_V v_2$) is a wfo, but not vice versa. The following shows how to obtain a wbr from a wfo. All missing proofs can be found in [24].

Lemma 1. (wbr from wfo) *Let $<_V$ be a well-founded order on V . Then \leq_V , defined by $v_1 \leq_V v_2$ iff $v_1 \not\prec_V v_2$, is a wbr on V . Furthermore, $<_V$ and \leq_V have the same set of admissible sequences.*

This means that, in an online setting, the approach based upon wbr's is in theory at least as powerful as the one based upon wfo's. Further below we will actually show that wbr's are strictly more powerful.

Observe that \leq_V is not necessarily a wqo: transitivity is not ensured as $t_1 \not\prec t_2$ and $t_2 \not\prec t_3$ do not imply $t_1 \not\prec t_3$. Let, e.g., $s < t$ denote that s is strictly more general than t . Then $<$ is a wfo (see below) but $p(X, X, a) \not\prec p(X, Z, b)$ and $p(X, Z, b) \not\prec p(X, Y, a)$ even though $p(X, X, a) > p(X, Y, a)$.

Let us now examine the power of one particular wqo, the earlier defined \preceq .

3.2 Homeomorphic Embedding and Monotonic wfo's

The homeomorphic embedding \preceq relation is very flexible. It will for example, when applied to the sequence of covering ancestors, permit the full unfolding of most terminating Datalog programs, the quicksort or even the mergesort program when the list to be sorted is known (the latter poses problems to some static termination analysis methods [38,27]; for some experiments see Appendix A). Also, the *produce-consume* example from [31] requires rather involved techniques (considering the context) to be solved by wfo's. Again, this example poses no problem to \preceq (cf. Appendix A). The homeomorphic embedding \preceq is also very powerful in the context of metaprogramming. Notably, it has the ability to “penetrate” layers of (non-ground) meta-encodings (cf. also Appendix A). For instance, \preceq will admit the following sequences (where Example 1 is progressively wrapped into “vanilla” metainterpreters counting resolution steps and keeping track of the selected predicates respectively):

Sequence
$rev([a, b T], [], R) \rightsquigarrow rev([b T], [a], R)$
$solve(rev([a, b T], [], R), 0) \rightsquigarrow solve(rev([b T], [a], R), s(0))$
$solve'(solve(rev([a, b T], [], R), 0), []) \rightsquigarrow solve'(solve(rev([b T], [a], R), s(0)), [rev])$

Again, this is very difficult for wfo's and requires refined and involved techniques (of which to our knowledge no implementation exists).

We have intuitively demonstrated the usefulness of \preceq and that it is often more flexible than wfo's. But can we prove some “hard” results? It turns out that we can and we now establish that — in the online setting — \preceq is strictly more generous than a large class of refined wfo's.

Definition 5. A well-founded order \prec on expressions is said to be monotonic iff the following rules hold:

1. $X \not\prec Y$ for all variables X, Y ,
2. $s \not\prec f(t_1, \dots, t_n)$ whenever f is a function symbol and $s \not\prec t_i$ for some i and
3. $f(s_1, \dots, s_n) \not\prec f(t_1, \dots, t_n)$ whenever $\forall i \in \{1, \dots, n\} : s_i \not\prec t_i$.

Note the similarity of structure with the definition of \leq (but, contrary to \leq , $\not\prec$ does not have to be the least relation satisfying the rules). This similarity of structure will later enable us to prove that any sequence admissible wrt \prec must also be admissible wrt \leq (by showing that $s \leq t \Rightarrow s \not\prec t$). Also observe that point 2 need not hold for predicate symbols and that point 3 implies that $c \not\prec c$ for all constant and proposition symbols c . Finally, there is a subtle difference between monotonic wfo's as of Definition 5 and wfo's which possess the replacement property (such orders are called rewrite orders in [37] and monotonic in [7]). More on that below.

Similarly, we say that a norm $\|\cdot\|$ (i.e. a mapping from expressions to \mathbb{N}) is said to be monotonic iff the associated wfo $\prec_{\|\cdot\|}$ is monotonic ($t_1 \prec_{\|\cdot\|} t_2$ iff $\|t_1\| < \|t_2\|$).

For instance the termsize norm (see below) is trivially monotonic. More generally, any semi-linear norm of the following form is monotonic:

Proposition 2. Let the norm $\|\cdot\| : Expr \rightarrow \mathbb{N}$ be defined by:

- $\|t\| = c_f + \sum_{i=1}^n c_{f,i} \|t_i\|$ if $t = f(t_1, \dots, t_n)$, $n \geq 0$
- $\|t\| = c_v$ otherwise (i.e. t is a variable)

Then $\|\cdot\|$ is monotonic if all coefficients $c_v, c_f, c_{f,i}$ are ≥ 0 and $c_{f,i} \geq 1$ for all function symbols f of arity ≥ 1 (but not necessarily for all predicate symbols).

Proof. As $<$ on \mathbb{N} is total we have that $s \not\prec t$ is equivalent to $s \leq t$. The proof proceeds by induction on the structure of the expressions and examines every rule of Definition 5 separately:

1. $X \leq Y$ for all variables X, Y
this trivially holds as we use the same constant c_v for all variables.
2. $s \leq f(t_1, \dots, t_n)$ whenever $s \leq t_i$ for some i
This holds trivially if all coefficients are ≥ 0 and if $c_{f,i} \geq 1$. This is verified, as the rule only applies if f is a function symbol.
3. $f(s_1, \dots, s_n) \leq f(t_1, \dots, t_n)$ whenever $\forall i \in \{1, \dots, n\} : s_i \leq t_i$
This holds trivially, independently of whether f is a function or predicate symbol, as all coefficients are positive (and the same coefficient is applied to s_i and t_i).

By taking $c_v = 0$ and $c_{f,i} = c_f = 1$ for all f we get the *termsize* norm $\|\cdot\|_{ts}$, which by the above proposition is thus monotonic. Also, by taking $c_v = 1$ and $c_{f,i} = c_f = 1$ for all f we also get a monotonic norm, counting symbols. Finally, a *linear norm* can always be obtained [38] by setting $c_v = 0$, $c_{f,i} = 1$ and $c_f \in \mathbb{N}$ for all f . Thus, as another corollary of the above, any linear norm is monotonic.

Proposition 3. Let $\|\cdot\|_1, \dots, \|\cdot\|_k$ be monotonic norms satisfying Proposition 2.

Then the lexicographical ordering \prec_{lex} defined by $s \prec_{lex} t$ iff

$\exists i \in \{1, \dots, k\}$ such that $\|s\|_i < \|t\|_i$ and $\forall j \in \{1, \dots, i-1\} : \|s\|_j = \|t\|_j$
is a monotonic wfo.

Proof. By standard results (see, e.g., [8]) we know that \prec_{lex} is a wfo (as $<$ is a wfo on \mathbb{N}). We will prove that \prec_{lex} satisfies all the rules of Definition 5.

1. First, rule 1 is easy as $\|X\|_i = \|Y\|_i$ for all i and variables X, Y and therefore we never have $X \prec_{lex} Y$.

2. Before examining the other rules, let us note that $s \not\prec_{lex} t$ is equivalent to saying that either

- a) $\forall j \in \{1, \dots, k\} \ \|s\|_j = \|t\|_j$ or
- b) there exists an $j \in \{1, \dots, k\}$ such that $\|s\|_j < \|t\|_j$ and $\forall l \in \{1, \dots, j-1\}$: $\|s\|_l = \|t\|_l$.

Let us now examine rule 2 of Definition 5. We have to prove that whenever $s \not\prec_{lex} t_i$ the conclusion of the rule holds.

Let us first examine case a) for $s \not\prec_{lex} t_i$. We have $\|s\|_j = \|t_i\|_j$ and thus we know that $\|s\|_j \leq \|f(t_1, \dots, t_n)\|_j$ by monotonicity of $\|\cdot\|_j$ (as $<$ on \mathbb{N} is total we have that $s \not\prec t$ is equivalent to $s \leq t$). As this holds for all $\|\cdot\|_j$ we cannot have $s_j \succ_{lex} f(t_1, \dots, t_n)$.

Let us now examine the second case b) for $s \not\prec_{lex} t_i$. Let $j \in \{1, \dots, k\}$ such that $\|s\|_j < \|t_i\|_j$ and $\forall l \in \{1, \dots, j-1\}$: $\|s\|_l = \|t_i\|_l$. For all l we can deduce as above that $\|s\|_l \leq \|f(t_1, \dots, t_n)\|_l$. However, we still have to prove that $\|s\|_j < \|f(t_1, \dots, t_n)\|_j$. By monotonicity of $\|\cdot\|_j$ we only know that $\|s\|_j \leq \|f(t_1, \dots, t_n)\|_j$. But we can also apply monotonicity of $\|\cdot\|_j$ to deduce that $\|t_i\|_j \leq \|f(t_1, \dots, t_n)\|_j$ and hence we can infer the desired property (as $\|s\|_j < \|t_i\|_j$).

3. Now, for rule 3 we have to prove that whenever $s_i \not\prec_{lex} t_i$ for all $i \in \{1, \dots, n\}$ the conclusion of the rule holds. There are again two cases.

a) We can have $\|s_i\|_j = \|t_i\|_j$ for all i, j . By monotonicity of each $\|\cdot\|_j$ we know that $\|f(s_1, \dots, s_n)\|_j \leq \|f(t_1, \dots, t_n)\|_j$ for all $j \in \{1, \dots, k\}$. Hence, we cannot have $f(s_1, \dots, s_n) \succ_{lex} f(t_1, \dots, t_n)$.

b) In the other case we know that there must be a value $j' \in \{1, \dots, k\}$ such that for some i : $\|s_i\|_{j'} < \|t_i\|_{j'}$ and $\forall l \in \{1, \dots, j'-1\}$: $\|s_i\|_l = \|t_i\|_l$. I.e., by letting j denote the minimum value j' for which this holds, we know that for some i : $\|s_i\|_j < \|t_i\|_j$ and for all i' : $\forall l \in \{1, \dots, j\}$: $\|s_{i'}\|_l \leq \|t_{i'}\|_l$. By monotonicity of each $\|\cdot\|_l$ we can therefore deduce that $\forall l \in \{1, \dots, j\}$: $\|f(s_1, \dots, s_n)\|_l \leq \|f(t_1, \dots, t_n)\|_l$. We can also deduce by monotonicity of $\|\cdot\|_j$ that $\|f(s_1, \dots, s_n)\|_j \leq \|f(t_1, \dots, t_n)\|_j$. We can even deduce that $\|f(s_1, \dots, s_n)\|_j \leq \|f(t_1, \dots, t_{i-1}, s_i, t_{i+1}, \dots, t_n)\|_j \leq \|f(t_1, \dots, t_n)\|_j$. Now, we just have to prove that:

$\|f(t_1, \dots, t_{i-1}, s_i, t_{i+1}, \dots, t_n)\|_j < \|f(t_1, \dots, t_n)\|_j$ in order to affirm that $\|f(s_1, \dots, s_n)\|_j \not\prec_{lex} \|f(t_1, \dots, t_n)\|_j$. This does not hold for all monotonic norms, but as we know that $\|\cdot\|_j$ satisfies Proposition 2, this can be deduced by the fact that the coefficient $c_{f,i}$ in $\|\cdot\|_j$ must be ≥ 1 .

It is important that the norms $\|\cdot\|_1, \dots, \|\cdot\|_k$ satisfy Proposition 2. Otherwise, a counterexample would be as follows. Let $\|a\|_1 = 1$, $\|b\|_1 = 2$ and $\|f(a)\|_1 = \|f(b)\|_1 = 5$. Also let $\|a\|_2 = 2$, $\|b\|_2 = 1$ and $\|f(a)\|_2 = 3$, $\|f(b)\|_2 = 2$. Now we have $a \prec_{lex} b$, i.e. $a \not\prec_{lex} b$, but also $f(a) \succ_{lex} f(b)$ and condition 3 of monotonicity for \prec_{lex} is violated.

One could make Proposition 3 slightly more general, but the current version is sufficient to show the desired result, namely that most of the wfo's used in online practice are actually monotonic. For example almost all of the refined wfo's defined in [5, 33, 32, 31] are monotonic:

- Definitions 3.4 of [5], 3.2 of [33] and 2.14 of [32] all sum up the number of function symbols (i.e. termsize) of a subset of the argument positions of

atoms. These wfo's are therefore immediately covered by Proposition 2. The algorithms only differ in the way of choosing the positions to measure. The early algorithms simply measure the input positions, while the later ones dynamically refine the argument positions to be measured (but which are still measured using the term-size norm).

- Definitions 3.2 of [32] as well as 8.2.2 of [31] use the lexicographical order on the term-sizes of some selected argument positions. These wfo's are therefore monotonic as a corollary to Propositions 2 and 3.

The only non-monotonic wfo in that collection of articles is the one defined specifically for metainterpreters in Definition 3.4 of [5] (also in Sect. 8.6 of [31]) which uses selector functions to focus on subterms to be measured. We will return to this approach below.

Also, as already mentioned, some of the techniques in [32,31] (in Sects. 3.4 and 8.2.4 respectively) do not require the *whole* sequence to be admissible wrt a unique wfo, i.e. one can split up a sequence into a (finite) number of subsequences and apply different (monotonic) wfo's on these subsequences. Similar refinements can also be developed for wqo's and the formal study of these refinements are (thus) not the main focus of the paper.

Before showing that \leq is strictly more powerful than the union of all monotonic wfo's, we adapt the class of simplification orderings from term rewriting systems. It will turn out that the power of this class is also subsumed by \leq .

Definition 6. *A simplification ordering is a wfo \prec on expressions which satisfies*

1. $s \prec t \Rightarrow f(t_1, \dots, s, \dots, t_n) \prec f(t_1, \dots, t, \dots, t_n)$ (replacement property),
2. $t \prec f(t_1, \dots, t, \dots, t_n)$ (subterm property) and
3. $s \prec t \Rightarrow s\sigma \prec t\gamma$ for all variable only substitutions σ and γ (invariance under variable replacement).

The third rule of the above definition is new wrt term-rewriting systems and implies that all variables must be treated like a unique new constant. It turns out that a lot powerful wfo's are simplification orderings [7,37]: recursive path ordering, Knuth-Bendix ordering or lexicographic path ordering, to name just a few. However, not all wfo's of Proposition 2 are simplification orderings: e.g., for $c_f = 0, c_a = 1$ we have $\|a\| = \|f(a)\|$ and the subterm property does not hold (for the associated wfo). In addition, Proposition 2 allows a special treatment for predicates. On the other hand, there are wfo's which are simplification orderings but are not monotonic according to Definition 5.

Proposition 4. *Let \prec be a wfo on expressions. Then any admissible sequence wrt \prec is also an admissible sequence wrt \leq if \prec is a) monotonic or if it is b) a simplification ordering.*

Proof. First, let us observe that for a given wfo \prec on expressions, any admissible sequence wrt \prec is also an admissible sequence wrt \leq iff $s \succ t \Rightarrow s \not\leq t$. Indeed (\Rightarrow), whenever $s \leq t$ then $s \not\succ t$, and this trivially implies (by transitivity of \prec) that any sequence not admissible wrt \leq cannot be strictly descending wrt \prec . On the other hand

(\Leftarrow), let us assume that for some s and t $s \trianglelefteq t$ but $s \succ t$. This means that the sequence s, t is admissible wrt \succ but not wrt \trianglelefteq and we have a contradiction.

a) The proof that for a monotonic wfo \prec we have $s \trianglelefteq t \Rightarrow s \not\succ t$ is by straightforward induction on the structure of s and t . The only “tricky” aspect is that the second rule for monotonicity only holds if f is a function symbol. But if f is a predicate symbol, then $s \trianglelefteq t_i$ cannot hold because we supposed that predicate and function symbols are distinct.

b) If \prec is a simplification ordering then we can apply Lemma 3.3 of [37] to deduce that \prec is the superset of the strict part of \trianglelefteq (i.e., $\prec \supseteq \trianglelefteq$). Let us examine the two possibilities for $s \trianglelefteq t$. First, we can have $s \triangleleft t$. In that case we can deduce $s \prec t$ and thus $s \not\succ t$. Second, we can have $s \trianglelefteq t$ and $t \trianglelefteq s$. In that case s and t are identical, except for the variables. If we now take the substitution σ which assigns all variables in s and t to a unique variable we have $s\sigma = t\sigma$, i.e., $s\sigma \not\succ t\sigma$. This means that $s \succ t$ cannot hold (because \succ is invariant under variable replacement).

This means that the admissible sequences wrt \trianglelefteq are a superset of the union of all admissible sequences wrt simplification orderings and monotonic wfo’s. In other words, no matter how much refinement we put into an approach based upon monotonic wfo’s or upon simplification orderings we can only expect to approach \trianglelefteq in the limit. But by a simple example we can even dispel that hope.

Example 3. Take the sequence $\delta = f(a), f(b), b, a$. This sequence is admissible wrt \trianglelefteq as $f(a) \not\trianglelefteq f(b)$, $f(a) \not\trianglelefteq b$, $f(a) \not\trianglelefteq a$, $f(b) \not\trianglelefteq b$, $f(b) \not\trianglelefteq a$ and $a \not\trianglelefteq b$. However, there is no monotonic wfo \prec which admits this sequence. More precisely, to admit δ we must have $f(a) \succ f(b)$ as well as $b \succ a$, i.e. $a \not\succ b$. But this violates rule 3 of Definition 5 and \prec cannot be monotonic. This also violates rule 1 of Definition 6 and \prec cannot be a simplification ordering.

These new results relating \trianglelefteq to monotonic wfo’s shed light on \trianglelefteq ’s usefulness in the context of ensuring online termination.

But of course the admissible sequences wrt \trianglelefteq are *not* a superset of the union of all admissible sequences wrt *any* wfo.¹ For instance the list-length norm $\|\cdot\|_{len}$ is not monotonic, and indeed we have for $t_1 = [1, 2, 3]$ and $t_2 = [[1, 2, 3], 4]$ that $\|t_1\|_{len} = 3 > \|t_2\|_{len} = 2$ although $t_1 \trianglelefteq t_2$. So there are sequences admissible wrt list-length but not wrt \trianglelefteq . The reason is that $\|\cdot\|_{len}$ in particular and non-monotonic wfo’s in general can completely ignore certain parts of the term, while \trianglelefteq will always inspect that part. E.g., if we have $s \succ f(\dots t \dots)$ and \succ ignores the subterm t then it will also be true that $s \succ f(\dots s \dots)$ while $s \trianglelefteq f(\dots s \dots)$ ² i.e. the sequence $s, f(\dots s \dots)$ is admissible wrt \succ but not wrt \trianglelefteq .

For that same reason the wfo’s for metainterpreters defined in Definition 3.4 of [5] mentioned above are not monotonic, as they are allowed to completely focus on subterms, fully ignoring other subterms. However, automation of that technique is not addressed in [5]. E.g., for this wfo one cannot immediately apply

¹ Otherwise \trianglelefteq could not be a wqo, as *all* finite sequences without repetitions are admissible wrt some wfo (map last element to 1, second last element to 2, ...).

² Observe that if f is a predicate symbols then $f(\dots s \dots)$ is not a valid expression, which enabled us to ignore arguments to predicates in e.g. Proposition 2.

the idea of continually refining the measured subterms, because otherwise one might simply plunge deeper and deeper into the terms and termination would not be ensured. A step towards an automatic implementation is presented in Sect. 8.6 of [31] and it will require further work to formally compare it with wqo-based approaches and whether the ability to completely ignore certain parts of an expression can be beneficial for practical programs. But, as we have seen earlier, \sqsubseteq alone is already very flexible for metainterpreters, even more so when combined with characteristic trees [26] (see also [46]).

Of course, for any wfo (monotonic or not) one can devise a wbr (cf. Lemma 11) which has the same admissible sequences. Still there are some feats that are easily attained, even by using \sqsubseteq , but which *cannot* be achieved by a wfo approach (monotonic or not). Take the sequences $S_1 = p([], [a]), p([a], [])$ and $S_2 = p([a], []), p([], [a])$. Both of these sequences are admissible wrt \sqsubseteq . This illustrates the flexibility of using well-quasi orders compared to well-founded ones in an online setting, as there exists *no* wfo (monotonic or not) which will admit *both* these sequences. It however also illustrates why, when using a wqo in that way, one has to compare with every predecessor state of a process. Otherwise one can get infinite derivations of the form $p([a], []) \rightarrow p([], [a]) \rightarrow p([a], []) \rightarrow \dots$ [3].

Short Note on Offline Termination. This example also shows why \sqsubseteq (or well-quasi orders in general) cannot be used *directly* for static termination analysis. Let us explain what we mean. Take, e.g., a program containing the clauses $C_1 = p([a], []) \leftarrow p([], [a])$ and $C_2 = p([], [a]) \leftarrow p([a], [])$. Then, in both cases the body is not embedding the head, but still the combination of the two clauses leads to a non-terminating program. However, \sqsubseteq can be used to *construct well-founded* orders for static termination analysis. Take the clause C_1 . The head and the body are incomparable according to \sqsubseteq . So, we can simply extend \sqsubseteq by stating that $p([a], []) \triangleright p([], a)$ (thus making the head strictly larger than the body atom). As already mentioned, for any extension \leq of a wqo we have that $<$ is a wfo. Thus we know that the program just consisting of C_1 is terminating. If we now analyse C_2 we have that, according to the extended wqo, the body is strictly larger than the head and (luckily) we cannot prove termination (i.e. there is no way of extending \sqsubseteq so that for both C_1 and C_2 the head is strictly larger than the body).

4 Discussion and Conclusion

Of course \sqsubseteq is not the ultimate relation for ensuring online termination. On its own in the context of local control of partial deduction, \sqsubseteq will sometimes allow

³ When using a wfo one has to compare only to the closest predecessor [32], because of the transitivity of the order and the strict decrease enforced at each step. However, wfo's are usually extended to incorporate variant checking and then require inspecting every predecessor anyway (though only when there is no strict weight decrease, see, e.g., [31, 32]).

too much unfolding than desirable for efficiency concerns (i.e. more unfolding does not always imply a better specialised program) and we refer to the solutions developed in, e.g., [26,18].

For some applications, \sqsubseteq remains too restrictive. In particular, it does not always deal satisfactorily with fluctuating structure (arising, e.g., for certain meta-interpretation tasks) [46]. The use of characteristic trees [23,26] remedies this problem to some extent, but not totally. A further step towards a solution is presented in [46]. In that light, it might be of interest to study whether the extensions of the homeomorphic embedding relation proposed in [39] and [21] (in the context of static termination analysis of term rewrite systems) can be useful in an online setting. As shown in [24] the treatment of variables of \sqsubseteq is rather rudimentary and several ways to remedy this problem are presented.

In summary, we have shed new light on the relation between wqo's and wfo's and have formally shown (for the first time) why wqo's are more interesting than wfo's for ensuring termination in an online setting (such as program specialisation or analysis). We have illustrated the inherent flexibility of \sqsubseteq and proved that, despite its simplicity, it is strictly more generous than the class of monotonic wfo's. As all the wfo's used for automatic online termination (so far) are actually monotonic, this formally establishes the interest of \sqsubseteq in that context.

Acknowledgements. I would like to thank Danny De Schreye, Robert Glück, Jesper Jørgensen, Bern Martens, Maurizio Proietti, Jacques Riche and Morten Heine Sørensen for all the discussions, comments and joint research which led to this paper. Anonymous referees as well as Bern Martens provided extremely useful feedback on this paper.

References

1. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialisation of lazy functional logic programs. In *Proceedings PEPM'97*, pages 151–162, Amsterdam, The Netherlands, 1997. ACM Press.
2. M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven partial evaluation of functional logic programs. In H. Riis Nielson, editor, *Proceedings ESOP'96*, LNCS 1058, pages 45–61. Springer-Verlag, 1996.
3. K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. North-Holland Amsterdam, 1990.
4. R. Bol. Loop checking in partial deduction. *The Journal of Logic Programming*, 16(1&2):25–46, 1993.
5. M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.
6. D. De Schreye and S. Decorte. Termination of logic programs: The never ending story. *The Journal of Logic Programming*, 19 & 20:199–260, May 1994.
7. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

8. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, MIT Press, 1990.
9. N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.
10. J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings PEPM'93*, pages 88–98. ACM Press, 1993.
11. R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S. Swierstra, editors, *Proceedings PLILP'96*, LNCS 1140, pages 152–166, Aachen, Germany, September 1996. Springer-Verlag.
12. R. Glück and M. H. Sørensen. A roadmap to supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 137–160, Schloß Dagstuhl, 1996. Springer-Verlag.
13. J. Gustedt. *Algorithmic Aspects of Ordered Structures*. PhD thesis, Technische Universität Berlin, 1992.
14. G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 2:326–336, 1952.
15. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.
16. N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.
17. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
18. J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Proceedings LOPSTR'96*, LNCS 1207, pages 59–82, Stockholm, Sweden, August 1996. Springer-Verlag.
19. J. B. Kruskal. Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.
20. L. Lafave and J. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In N. Fuchs, editor, *Proceedings LOPSTR'97*, LNCS, Leuven, Belgium, July 1997. Springer-Verlag.
21. P. Lescanne. Well rewrite orderings and well quasi-orderings. Technical Report N° 1385, INRIA-Lorraine, France, January 1991.
22. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. 1996–1998. Obtainable via <http://www.cs.kuleuven.ac.be/~dtai>.
23. M. Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, K.U. Leuven, May 1997. Accessible via <http://www.cs.kuleuven.ac.be/~michael>.
24. M. Leuschel. Homeomorphic Embedding for Online Termination. Technical Report, Department of Electronics and Computer Science, University of Southampton, 1998.
25. M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996. Springer-Verlag.
26. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.

27. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. Unfolding the mystery of mergesort. In N. Fuchs, editor, *Proceedings LOPSTR'97*, LNCS, Leuven, Belgium, July 1997. Springer-Verlag.
28. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
29. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.
30. R. Marlet. *Vers une Formalisation de l'Évaluation Partielle*. PhD thesis, Université de Nice - Sophia Antipolis, December 1994.
31. B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. PhD thesis, K.U. Leuven, February 1994.
32. B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996.
33. B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122(1–2):97–117, 1994.
34. B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613, Kanagawa, Japan, June 1995. MIT Press.
35. J. Martin. Sonic partial deduction. Technical Report, Department of Electronics and Computer Science, University of Southampton, 1998.
36. P.-A. Melliès. On a duality between Kruskal and Dershowitz theorems. In K. G. Larsen, editor, *Proceedings of ICALP'98*, LNCS, Aalborg, Denmark, 1998. Springer-Verlag.
37. A. Middeldorp and H. Zantema. Simple termination of rewrite systems. *Theoretical Computer Science*, 175(1):127–158, 1997.
38. L. Plümer. *Termination Proofs for Logic Programs*. LNCS 446. Springer-Verlag, 1990.
39. L. Puel. Using unavoidable set of trees to generalize Kruskal's theorem. *Journal of Symbolic Computation*, 8:335–382, 1989.
40. E. Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, March 1993.
41. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
42. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Proceedings ILPS'95*, pages 465–479, Portland, USA, December 1995. MIT Press.
43. M. H. Sørensen, R. Glück and N.D. Jones. A positive supercompiler. *Journal of Functional Programming*, 1996.
44. J. Stillman. *Computational Problems in Equational Theorem Proving*. PhD thesis, State University of New York at Albany, 1988.
45. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
46. W. Vanhoof and B. Martens. To parse or not to parse. In N. Fuchs, editor, *Proceedings LOPSTR'97*, LNCS, Leuven, Belgium, July 1997. Springer-Verlag.
47. D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architectures*, LNCS 523, pages 165–191, Harvard University, 1991. Springer-Verlag.

A Small Experiments with the ECCE System

The purpose of this appendix is to illustrate the flexibility which the homeomorphic embedding relation provides straight “out of the box” (other more intricate well-quasi orders, like the one used by Mixtus [41], can handle some of the examples below as well).

For that we experiment with the ECCE partial deduction system [22] using an unfolding rule based on \sqsubseteq which allows the selection of either determinate literals or left-most literals within a goal, given that no covering ancestor [5] is embedded (via \sqsubseteq). To ease readability, the specialised programs are sometimes presented in unrenamed form.

First, let us take the *mergesort* program, which is somewhat problematic for a lot of static termination analysis methods [38,27].

```
mergesort([], []).
mergesort([X], [X]).
mergesort([X,Y|Xs], Ys) :-
    split([X,Y|Xs], X1s, X2s),
    mergesort(X1s, Y1s), mergesort(X2s, Y2s),
    merge(Y1s, Y2s, Ys).

split([], [], []).
split([X|Xs], [X|Ys], Zs) :- split(Xs, Zs, Ys).

merge([], Xs, Xs).
merge(Xs, [], Xs).
merge([X|Xs], [Y|Ys], [X|Zs]) :- X =< Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X>Y, merge([X|Xs], Ys, Zs).
```

The partial evaluation query:

```
?- mergesort([3,1,2], X).
```

As the following resulting specialised program shows, homeomorphic embedding allowed the full unfolding of *mergesort*:

```
mergesort([3,1,2], [1,2,3]).
```

It took ECCE less than 0.5 s on a Sparc Classic to produce the above program (including post-processing and writing to file).

We can even achieve this same feat even if we interpose one or more levels of metainterpretation! Take a vanilla solve metainterpreter with *mergesort* at the object-level:

```
solve([]).
solve([H|T]) :-
    claus(H, Bdy), solve(Bdy), solve(T).

claus(mergesort([], []), []).
claus(mergesort([X], [X]), []).
```

```

claus(mergesort([X,Y|Xs],Ys),
      [split([X,Y|Xs],X1s,X2s),
       mergesort(X1s,Y1s),mergesort(X2s,Y2s),
       merge(Y1s,Y2s,Ys) ] ).
claus(split([],[],[]), []).
claus(split([X|Xs],[X|Ys],Zs) , [ split(Xs,Zs,Ys) ] ).
claus(merge([],Xs,Xs), []).
claus(merge(Xs,[],Xs), []).
claus(merge([X|Xs],[Y|Ys],[X|Zs]) ,
      [ X =< Y, merge(Xs,[Y|Ys],Zs) ] ).
claus(merge([X|Xs],[Y|Ys],[Y|Zs]) ,
      [ X > Y, merge([X|Xs],Ys,Zs)] ).
claus('=<'(X,Y),[]) :- X =< Y.
claus('>'(X,Y),[]) :- X > Y.

```

```
mergesort_test(X) :- solve([mergesort([3,1,2],X)]).
```

The partial evaluation query:

```
?- mergesort_test(X).
```

Again homeomorphic embedding allowed the full unfolding:

```
mergesort_test([1,2,3]).
```

It took ECCE 2.86 s on a Sparc Classic to produce the above program (including post-processing and writing to file).

The following example is taken from [31].

```

produce([], []).
produce([X|Xs],[X|Ys]) :- produce(Xs,Ys).

consume([]).
consume([X|Xs]) :- consume(Xs).

```

The partial evaluation query:

```
?- produce([1,2|X],Y), consume(Y).
```

To solve it in the setting of unfolding based upon wfo's one needs both partition based measure functions as well as taking the context into account. The same adequate unfolding can simply be achieved by \leq based determinate unfolding, as illustrated by the following specialised program derived by ECCE (default settings):

```

produce_conj__1([], [1,2]).
produce_conj__1([X1|X2], [1,2,X1|X3]) :-
  produce_conj__2(X2,X3).
produce_conj__2([], []).
produce_conj__2([X1|X2], [X1|X3]) :-
  produce_conj__2(X2,X3).

```

Analysis of Imperative Programs through Analysis of Constraint Logic Programs

Julio C. Peralta, John P. Gallagher, and Hüseyin Sağlam

University of Bristol
Dept. of Computer Science
Merchant Venturers Building
Woodland Rd., Bristol, U.K. BS8 1UB
fax: +44-117-9545208

e-mail: {jperalta, john}@cs.bris.ac.uk, saglam@osf02.ktu.edu.tr

Abstract. In this paper a method is proposed for carrying out analysis of imperative programs. We achieve this by writing down the language semantics as a declarative program (a constraint logic program, in the approach shown here). We propose an effective style of writing operational semantics suitable for analysis which we call *one-state small-step* semantics. Through controlled partial evaluation we are able to generate residual programs where the relationship between imperative statements and predicates is straightforward. Then we use a static analyser for constraint logic programs on the residual program. The analysis results are interpreted through program points associating predicates in the partially evaluated interpreter to statements in its corresponding imperative program. We used an analyser that allows us to determine linear equality, inequality and disequality relations among the variables of a program without user-provided inductive assertions or human interaction. The proposed method intends to serve as a framework for the analysis of programs in any imperative language. The tools required are a partial evaluator and a static analyser for the declarative language.

Keywords: Partial Evaluation, Constraint Logic Programming, Operational Semantics, Imperative Program Analysis.

1 Introduction

Program semantics has long been used as a formal basis for program manipulation. By this we mean that the formal semantics of a programming language is written down in some mathematical framework, which is then used to establish program properties such as termination, correctness with respect to specifications, or the correctness of program transformations.

In the case of imperative languages the gap between semantics and programs is greater than in the case of declarative languages. Perhaps for this reason, semantics-based tools for declarative languages, such as abstract interpreters and partial evaluators, have been the subject of more intense research than similar tools for imperative languages.

The aim of this paper is to show how to transfer results achieved in declarative languages to imperative languages. The approach is to implement the semantics of imperative languages carefully in a declarative language for which analysis and transformation tools exist.

There exist several kinds of semantics for imperative languages. The choice of which one is better suited for the particular application is a subject of ongoing research [3]. We shall see that for the purposes of this paper a variant of operational semantics will suffice.

Logic programming appears to be an elegant paradigm for imperative language semantics implementation. When written carefully, the semantics can be regarded as an *interpreter* for the imperative language. Appropriate techniques for implementing semantics are discussed in Sect. 2. Partial evaluation of the interpreter with respect to an imperative program yields an equivalent declarative program. By so doing we open up the possibility of applying well-developed techniques for analysis and transformation of constraint logic programs to imperative programs as well. Nevertheless, it is not clear how to relate the results of such analysis and/or transformation back to the original imperative program.

In order to obtain a correspondence between the imperative program and its corresponding declarative program some tuning of the partial evaluator is needed. Otherwise, the partial evaluator may remove important information needed to relate imperative statements and variables with their declarative counterpart. Such tuning involves selecting among the predicates of the semantics-based interpreter those we want to be defined in the residual program. Hence, we choose predicates from the semantics-based interpreter that relate directly to the meaning of the statements in the imperative program to be partially evaluated. As a result we get one predicate for each statement of the imperative program, thus highlighting the correspondence between imperative statements and predicates in the residual program.

In this paper we propose a method that intends to serve as a framework for the analysis of programs in any imperative language, by writing down its semantics as a declarative program (a constraint logic program, in the approach shown here). The tools required are a partial evaluator and a static analyser for the declarative language.

Section 2 considers the overall problem of encoding semantics in a logic programming language, in a form suitable for analysis. Section 3 provides some remarks on the implementation of the operational semantics for a small imperative language¹. In Sect. 4 we show the partial evaluation process and give an example. Section 5 illustrates how to relate the results of the analysis back to its imperative program in a systematic way. Finally, in Sects. 6 and 7 we discuss related work, and state our final remarks and some trends for future work.

¹ Currently we are writing the semantics for Java in a similar style

2 Analysis through Semantics

Before describing our experiments in detail let us consider some critical points concerning representing semantics in a form suitable for analysis. There are several styles of semantics for imperative languages to be found in textbooks. These may all be translated more or less directly into declarative programming languages, but it is necessary to consider carefully the choice of semantics and the style in which the semantics is represented. The choice is influenced by two questions: firstly, what kind of analysis is to be performed on the imperative program, and secondly, how can the complexity of the analysis be minimised?

2.1 Big-Step and Small-Step Semantics

The usual distinction between different kinds of semantics is between the compositional and operational styles. However for our purpose, the most relevant division is between *big-step* and *small-step* semantics. Note that operational semantics can be either big-step (natural semantics) or small-step (structural operational semantics).

Let us represent program statements by S and computation states by E . Big-step semantics is modelled using a relation of the form $bigstep(S, E_1, E_2)$, which means that the statement S transforms the initial state E_1 to final state E_2 . The effect of each program construct is defined independently; compound statements are modelled by composing together the effects of its parts.

On the other hand, small-step semantics is typically modelled by a transition relation of the form $smallstep(\langle S_1, E_1 \rangle, \langle S_2, E_2 \rangle)$. This states that execution of statement S_1 in state E_1 is followed by the execution of statement S_2 in state E_2 . Small-step semantics models a computation as a sequence of computation states, and the effect of a program construct is defined as its effect on the computation state in the context of the program in which it occurs.

Big steps can be derived from small steps by defining a special terminating statement, say *halt*, and expressing big-step relations as a sequence of small steps.

$$\begin{aligned} bigstep(S, E_1, E_2) &\leftarrow smallstep(\langle S, E_1 \rangle, \langle halt, E_2 \rangle). \\ bigstep(S, E_1, E_3) &\leftarrow smallstep(\langle S, E_1 \rangle, \langle S_1, E_2 \rangle), bigstep(S_1, E_2, E_3). \end{aligned}$$

For the purposes of program analysis, the two styles of semantics have significant differences. Analysis of the *bigstep* relation allows direct comparison of the initial and final states of a computation. As shown above, big steps are derivable from small steps but the analysis becomes more complex. If the purpose of analysis is to derive relationships between initial and final states, then big-step semantics would be recommended.

On the other hand, the *smallstep* relation represents directly the relation between one computation state and the next, information which would be awkward (though not impossible) to extract from the big-step semantics. Small-step semantics is more appropriate for analyses where local information about states is required, such as the relationship of variables within the same state, or between successive states.

2.2 One-State and Two-State Semantics

There is another option for representing small-step semantics, which leads to programs significantly simpler to analyse. Replace the *smallstep*($\langle S_1, E_1 \rangle, \langle S_2, E_2 \rangle$) relation by a clause $exec(S_1, E_1) \leftarrow exec(S_2, E_2)$. The meaning of the predicate $exec(S, E)$ is that there exists a terminating computation of the statement S starting in the state E . We also have to add a special terminal statement called *halt* which is placed at the end of every program, and a statement $exec(halt, E) \leftarrow true$.

We call this style of small-step semantics a *one-state* semantics since the relation *exec* represents only one state, in contrast to *two-state* semantics given by the *bigstep* and *smallstep* relations.

As an aside, the one-state and two-state styles of representation follow a pattern identified by Kowalski in [10], when discussing graph-searching algorithms in logic programming. Kowalski noted that there were two ways to formalise the task of searching for a path from node a to node z in a directed graph. One way is to represent the graph as a set of facts of the form $go(X, Y)$ representing arcs from node X to node Y . A relation $path(X, Y)$ could then be recursively defined on the arcs. The search task is to solve the goal $\leftarrow path(a, z)$. Alternatively, an arc could be represented as a clause of form $go(Y) \leftarrow go(X)$. In this case, to perform the task of searching for a path from node a to node z , the fact $go(a) \leftarrow true$ is added to the program, and computation is performed by solving the goal $\leftarrow go(z)$. There is no need for a recursively defined path relation, since the underlying logic of the implication relation fulfils the need.

One-state semantics corresponds to the use of the relation $go(X)$ while two-state semantics corresponds to using the relation $go(X, Y)$. The “start state” corresponds to the clause $exec(halt, E) \leftarrow true$. Our experiments show that the one-state semantics is considerably simpler to analyse than the two-state semantics.

2.3 Analysis of the Semantics

It may be asked whether one-state semantics is expressive enough, since no output state can be computed. That is, given a program P and initial state E , the computation is simulated by running the goal $\leftarrow exec(P, E)$ and the final state is not observable. This is certainly inadequate if we are using our semantics to simulate the full effect of computations.

However, during program analysis of $\leftarrow exec(P, E)$ more things are observable than during normal execution of the same goal. In particular we can use a program analysis algorithm that gives information about calls to different program subgoals. In one-state semantics with a relation $exec(S, E)$, the analysis can determine (an approximation of) every instance of $exec(S, E)$ that arises in the computation. We can even derive information about the relation of successive states since the analysis can derive information about instances of a clause $exec(S_1, E_1) \leftarrow exec(S_2, E_2)$.

In the experiments to be described below, we start from a structural operational semantics in a textbook style, and derive a one-state small-step semantics.

3 A Semantics-Based Interpreter

As already mentioned a desirable property of structural operational semantics is the way it reflects every change in the computation state. Here we present briefly a way of systematically translating formal operational semantics (adapted from [16]) into a constraint logic program.

We shall first provide some elements of the syntax of the imperative language and the metavariables used in the semantics descriptions. Assume we have an imperative language L with assignments, arithmetic expressions, while statements, if-then-else conditionals, empty statement, statement composition², and boolean expressions. Let S be a statement, a be an arithmetic expression, b a boolean expression, e a variable environment (mapping variables to their value), and x a program variable. All these variables may occur subscripted. A pair $\langle S, e \rangle$ is a *configuration*. Also a state is a special terminal configuration. The operational semantics below give meaning to programs by defining an appropriate transition relation holding between configurations.

3.1 Structural Operational Semantics

Using structural operational semantics [16] a transition relation \Rightarrow defines the relationship between successive configurations. There are different kinds of transition corresponding to different kinds of statement. Accordingly, the meaning of empty statement, assignment statement, if-then-else statement, while-do statement and composition of statements are:

$$\langle \text{skip}, e \rangle \Rightarrow e \quad (1)$$

$$\langle x := a, e \rangle \Rightarrow e[x \mapsto \mathcal{A}[a]e] \quad (2)$$

$$\langle \text{if } b \text{ then } S_1 \text{ else } S_2, e \rangle \Rightarrow \langle S_1, e \rangle \text{ if } \mathcal{B}[b]e = \mathbf{tt} \quad (3)$$

$$\langle \text{if } b \text{ then } S_1 \text{ else } S_2, e \rangle \Rightarrow \langle S_2, e \rangle \text{ if } \mathcal{B}[b]e = \mathbf{ff} \quad (4)$$

$$\langle \text{while } b \text{ do } S, e \rangle \Rightarrow \langle \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}, e \rangle \quad (5)$$

$$\langle S_1; S_2, e \rangle \Rightarrow \langle S'_1; S_2, e' \rangle \text{ if } \langle S_1, e \rangle \Rightarrow \langle S'_1, e' \rangle \quad (6)$$

$$\langle S_1; S_2, e \rangle \Rightarrow \langle S_2, e' \rangle \text{ if } \langle S_1, e \rangle \Rightarrow e' \quad (7)$$

where $\mathcal{A}[\cdot]$ is the semantics function for arithmetic expressions and $\mathcal{B}[\cdot]$ is the semantics function for boolean expressions. Intuitively, the assignment axiom schema above says that in a state e , $x := a$ is executed to yield a state $e[x \mapsto \mathcal{A}[a]e]$ which is as e except that x has the value $\mathcal{A}[a]e$. Moreover, transition 6 expresses that if S_1 is not a primitive statement of the language then execution won't proceed to S_2 until the rest of S_1 , S'_1 , has been fully executed. Transition 7 considers the case when execution of S_1 has been completed thus yielding state e' , hence execution of S_2 starts from this new state.

We may specialise transitions 6 and 7 by unfolding their conditions with respect to transitions 1, 2, 3, 4, and 5 above to transitions 9, 10, 11, 12, and 13

² The statement composition operator ';' is assumed to be right associative

below. Transition 8 below is obtained by unfolding the condition of transition 6 with respect to itself and applying the associativity property of the ‘;’ operator. We assume that cases 2, 3, 4 and 5 do not occur since all programs are terminated by *skip*. Hence the new semantics:

$$\langle \text{skip}, e \rangle \Rightarrow e$$

$$\langle (S_1; S_2); S_3, e \rangle \Rightarrow \langle S_1; (S_2; S_3), e \rangle \quad (8)$$

$$\langle \text{skip}; S_2, e \rangle \Rightarrow \langle S_2, e \rangle \quad (9)$$

$$\langle x := a; S_2, e \rangle \Rightarrow \langle S_2, e[x \mapsto \mathcal{A}[a]e] \rangle \quad (10)$$

$$\langle (\text{if } b \text{ then } S_1 \text{ else } S_2); S_3, e \rangle \Rightarrow \langle S_1; S_3, e \rangle \text{ if } \langle \mathcal{B}[b]e = \mathbf{tt} \rangle \quad (11)$$

$$\langle (\text{if } b \text{ then } S_1 \text{ else } S_2); S_3, e \rangle \Rightarrow \langle S_2; S_3, e \rangle \text{ if } \langle \mathcal{B}[b]e = \mathbf{ff} \rangle \quad (12)$$

$$\langle (\text{while } b \text{ do } S); S_2, e \rangle \Rightarrow \quad (13)$$

$$\langle \text{if } b \text{ then } (S; \text{while } b \text{ do } S \text{ else skip}); S_2, e \rangle$$

Next, we express the semantics as a constraint logic program below. It is worth noting that this representation aids the analysis phase by carrying a single environment instead of double environment (that is, the one-state small-step semantics discussed in Sect. 2).

```

exec(skip,E) <-
exec(compose(compose(S1,S2),S3),E) <-
    exec(compose(S1,compose(S2,S3)),E)
exec(compose(skip,S2),E) <- exec(S2,E)
exec(compose(assign(X,Ae),S2),E) <- a_expr(E,Ae,V),
                                     update(E,E2,X,V),
                                     exec(S2,E2)
exec(compose(ifte(B,S1,S2),S3),E) <- b_expr(E,B,true),
                                     exec(compose(S1,S3),E)
exec(compose(ifte(B,S1,S2),S3),E) <- b_expr(E,B,false),
                                     exec(compose(S2,S3),E)
exec(compose(while(B,S1),S2),E) <-
    exec(compose(ifte(B,compose(S1,while(B,S1)),skip),skip),E)

```

In this program the term `assign(X,A)` represents an assignment statement $X := A$, and the term `compose(S1,S2)` represents the composition of statement S_1 with statement S_2 . The predicate `exec(P,E)` holds iff program P can be executed from state E . The predicate `update(E1,E2,X,V)` models the change of state from E_1 to E_2 induced by the assignment of V to X . The value of the assigned variable X is changed to V but all other values of variables are passed unchanged from E_1 to E_2 . Observe that the clause for composition of statements in this implementation has been specialised to the individual cases of our program constructs, namely `assign`, `skip`, `ifte` and `while` in this example. Therefore we do not use the clauses corresponding to composition of statements in their general form.

One way to enhance the partial evaluation and the analysis of the semantics-based interpreter is to add constraints in the definition of predicates that handle arithmetic. For instance, define `a_expr` for addition of arithmetic expressions as:

```
a_expr(E,plus(Ae1,Ae2),V) <- a_expr(E,Ae1,V1), a_expr(E,Ae2,V2),
                               {V=V1+V2}
```

where the curly brackets are used to invoke the constraint solver. This enables constraint-based techniques to be applied [18], [20].

4 Partial Evaluation

Partial evaluation generates a *residual program* by *evaluating* a program with *part* of its input which is available at compile-time. When given the remaining part of the input, the residual program produces the same output as the source program when presented with the whole input. The aim of partial evaluation is to gain efficiency by specialising the source program, exploiting the known input.

A partial evaluator is applied to our semantics-based interpreter with respect to an input imperative program. The result is a residual logic program which is a version of the semantics interpreter specialised with respect to the input program. There is a well known relationship between partial evaluation and compilation, and the residual program can be viewed as the result of compiling the imperative program into the declarative language in which the semantics interpreter is written (a constraint logic program in our case). The residual program is then analysed, and the analysis results are related to the original imperative program, as we will discuss.

The role of partial evaluation is two-fold. One is to increase the efficiency and precision of the analysis. The other is to allow the results of the analysis to be mapped directly to the structure of the original imperative program. Analysis of imperative programs commonly associates a set of assertions with program points occurring between statements. By contrast, our top-down analyser for constraint logic programs produces a separate analysis per program predicate. This suggests having a predicate in the residual program corresponding to each program point in the imperative program.

Thus, for the purposes of analysis, partial evaluation is not required to produce the maximum possible specialisation. It suffices to have a translator from imperative to logic programs where imperative statements and variables can be explicitly related to predicates and their arguments. Some special purpose control over standard partial evaluation is required, which will be described below, but first we need to describe the data structure which is used to represent the imperative program.

4.1 Representation of Imperative Programs

Starting from the standard textual representation of imperative programs the first step is to produce a list of tokens. As the next step in producing the desired

specialisation we add a unique label to each statement of the input imperative program. These labels are used in the control of the partial evaluation. The term representation is produced using a conventional LALR parser [1]. The parser takes for input the tokenised representation of the imperative program and produces as output a list of terms ³ suitable for interpretation by the semantics-based interpreter. Such a list of terms contain labels that will allow us to identify every statement in the imperative program. For instance, consider the following program fragment representing an if-then-else statement with one assignment in each conditional branch:

```
[if,a,>,b,then,
  x, :=, a, +, 2, ;,
else,
  y, :=, b, -, 1]
```

By parsing the above code we obtain:

```
[ifte(gt(a,b),
  [assign(x,plus(a,2))],
  [assign(y,minus(b,1))])]
```

When decorated with program points we get:

```
[p(1,ifte(gt(a,b),
  p(2,[assign(x,plus(a,2))]),
  p(3,[assign(y,minus(b,1))])
)))]
```

We use the function symbol `p` to enclose a program point. The first argument of `p` is a program point label and the second its associated program statement.

The semantics-based interpreter is modified to handle the newly extended representation of statements. For example, instead of

```
exec(compose(assign(X,Ae),S2),E1) <- a_expr(E1,Ae,V),
                                     update(E1,E2,X,V),
                                     exec(S2,E2)
```

we write

```
exec(compose(p(La,assign(X,Ae)),S2),E1) <- a_expr(E1,Ae,V),
                                             update(E1,E2,X,V),
                                             exec(S2,E2)
```

A similar modification is made for every semantics clause defining a program statement and its program point.

³ For simplicity we use the list function symbol and Prolog list notation instead of the `compose` function symbol to denote composition of statements. Instead of `compose(S1,S2)` we write `[S1|S2]`.

4.2 Control of Partial Evaluation

Standard partial evaluation of logic programs [12], given a query and a program, seeks *a set of atoms* for which two properties hold, closedness and independence. To achieve both goals (closedness and independence) partial evaluators usually distinguish two levels of control:

- the *global control*, in which one decides which atoms will be partially evaluated, and
- the *local control*, in which one constructs the finite partial computation trees for each individual atom in the set and thus determines what the definitions of the partially evaluated atoms look like.

The global control determines whether the set of atoms contains more than one version of each predicate (*polyvariant control*) or whether only one version of every predicate is kept (*monovariant control*). We require polyvariant control since each program point should result in a distinct predicate in the residual program; for instance, a version of the semantics transition dealing with assignment statements should be produced for each assignment in the input program.

During partial evaluation global and local control interact, passing information between each other. For our purposes, we require the local control to evaluate the parts of the interpreter that break down the imperative program into its constituent statements (labelled program points).

The algorithm for partial evaluation that we use is based on the basic algorithm presented in [4], where we define the local control rule **Cr** and the abstraction operation **abstract** to cater for program points.

INPUT: a program *P* and goal *G*, and local control rule **Cr**.

OUTPUT: a set of atoms

```

begin
  A[0] := the set of atoms in G
  i := 0
  repeat
    P' := a partial evaluation of A[i] in P using Cr
    R := A[i] U { p(t) | B <- Q, p(t), Q' in P' OR
                                     B <- Q, not(p(t)), Q' in P' }
    A[i+1] := abstract(R)
    i := i+1
  until A[i] = A[i-1] (modulo variable renaming)
end

```

The global control is provided by the operation **abstract**(*S*). Our partial evaluator uses characteristic trees [6] as an aid in controlling the polyvariance and keeping the set of atoms finite at the global control level. The characteristic tree of an atom is a term capturing the shape of the computation tree produced for that atom using **Cr**.

Global control based on characteristic trees is modified to include the program point labels. The abstraction operation uses the following algorithm

- Let R be a finite set of atoms. Let R_{label} be the set of atoms in R which contain an argument of the form $p(N, Stm)$, where Stm is one of **assign**, **ifte** or **while**, and N is a program point label. Let R_{chtree} be the remaining atoms in R .
- Let $\{R_{N_1}, \dots, R_{N_k}\}$ be the finite partition of R_{label} where all atoms in R_{N_j} contain the argument $p(N_j, Stm)$.
- Let $\{R_{c_1}, \dots, R_{c_m}\}$ be the finite partition of R_{chtree} where R_{c_i} is the set of atoms in R_{chtree} having characteristic tree c_i .
- Let $abstract(R) = \{msg(R_{N_1}), \dots, msg(R_{N_k}), msg(R_{c_1}), \dots, msg(R_{c_m})\}$.

This is a correct abstraction according to the conditions in [4]. The purpose of the abstraction is to preserve a separate atom for each predicate that manipulates a program point, and to use characteristic tree abstraction for the other atoms. Note that the most specific generalisation (*msg*) preserves the program point argument.

At the local control level we use a determinate unfolding rule that generates SLDNF-trees with shower shape in the sense of [11, p. 36], suspending the evaluation whenever the semantics handles a program point. Some polyvariance and thus specialisation can be lost compared to an abstraction based on characteristic trees alone, but the result is appropriate for the needs of the analysis.

Example. Consider the contrived imperative program

```

(1)  i := 2;
(2)  j := 0;
(3)  while (n*n > 1) do
(4)    if (n*n = 2) then
(5)      i := i+4;
      else
(6)      i := i+2;
(7)      j := j+1;
    endwhile

```

By partial evaluation with respect to the above program and unknown initial environment we obtain:

<pre> program(X1,X2,X3) <- assign_1(X1,X2,X3) assign_1(X1,X2,X3) <- assign_2(X2,X3) assign_2(X1,X2) <- loop_1(2,0,X2) loop_1(X1,X2,X3) <- X4 is X3*X3, gt_test_1(X4,X5), do_1(X1,X2,X3,X5) gt_test_1(X1,tt) <- X1>1 </pre>	<pre> cond_1(X1,X2,X3) <- X4 is X3*X3, eq_test_1(X4,X5), cnd_1(X1,X2,X3,X5) cnd_1(X1,X2,X3,tt) <- assign_6(X1,X2,X3) cnd_1(X1,X2,X3,ff) <- assign_7(X1,X2,X3) assign_6(X1,X2,X3) <- X4 is X1+4, loop_1(X4,X2,X3) </pre>
--	---

```

gt_test_1(X1,ff) <- X1=<1
eq_test_1(X1,tt) <- X1>=2
eq_test_1(X1,ff) <- X1=\=2
do_1(X1,X2,X3,ff) <-
do_1(X1,X2,X3,tt) <-
    cond_1(X1,X2,X3)
assign_7(X1,X2,X3) <-
    X4 is X1+2,
    assign_8(X4,X2,X3)
assign_8(X1,X2,X3) <-
    X4 is X2+1,
    loop_1(X1,X4,X3)

```

where the predicate name prefixes `assign`, `cond`, and `loop` denote assignment, if-then-else, and while statements respectively. Here the correspondence between predicates and program points is as follows; (this information can be extracted automatically from the partial evaluator).

Predicate	Program point	Predicate	Program point
<code>assign_1</code>	(1)	<code>assign_6</code>	(5)
<code>assign_2</code>	(2)	<code>assign_7</code>	(6)
<code>loop_1</code>	(3)	<code>assign_8</code>	(7)
<code>cond_1</code>	(4)		

5 Results

Decorating statements in the imperative program with unique identifiers leads to ways of relating the imperative program to a residual CLP program. That is, for every program point of the imperative program we generate a clause head in the corresponding residual program thus providing information of how the specialised CLP program relates to the imperative program. Then by using logic program analysis tools we can obtain information from the CLP residual program. Through the program points it is straightforward to translate the results of the CLP program analysis to an imperative program analysis.

Example (continued): Once we have an appropriate constraint logic program representing an imperative program with the desired clause heads we input this program to the analyser for constraint logic programs [18]. By analysing the residual program shown at the end of Sect. 4.2 we obtain the following results:

```

program_query(X1,X2,X3).
assign_1_query(X1,X2,X3).
assign_2_query(X1,X2).
assign_6_query(X1,X2,X3) :-X2>=0.0,X2-0.5*X1=< -1.0.
loop_1_query(X1,X2,X3) :-X2>=0.0,X2-0.5*X1=< -1.0.
cond_1_query(X1,X2,X3) :-X2>=0.0,X2-0.5*X1=< -1.0.
assign_7_query(X1,X2,X3) :-X2>=0.0,X2-0.5*X1=< -1.0.
assign_8_query(X1,X2,X3) :-X2>=0.0,X2-0.5*X1=< -2.0.

```

where we have a set of constraints associated with each predicate in the analysed program⁴. The suffix `_query` in every predicate name indicates that the

⁴ We show the results only for the relevant predicates

constraints hold for the variables named every time we call that predicate during program execution. The query predicates are produced by query-answer transformation [7] applied to the partially evaluated program and the initial goal. This transformation takes a program and a goal and constructs definitions of the subgoals and their answers that arise during the computation of the goal.

Since we have a clause defining a predicate associated with a program point we may then say that the constraints associated with a predicate query are those that hold before the program point to which that predicate refers.

Once we have identified which is the relationship holding between the predicates and the program points it remains to determine how variables relate.

Partial evaluation performs some transformations regarding constant arguments and multiple occurrence of the same variable in an atom, when constructing the residual program. *Argument filtering and flattening* [5] are applied by our partial evaluator. For the purposes of our analysis we have suppressed them because they obscure the correspondence between logic programs and their imperative counterpart. For instance, the clause head for the predicate `assign_6` in the last example of Sect. 4.2 without redundant argument filtering and flattening is:

```
assign_6(5, [[i,j,n],[X1,X2,X3]], i, plus(i,4),
[p(3,while(gt(times(n,n),1),
  [p(4,ifte(eq(times(n,n),2),
    [p(5,assign(i,plus(i,4)))],
    [p(6,assign(i,plus(i,2)))],
    p(7,assign(j,plus(j,1)))]
  )]]))]).
```

The application of both these transformations to the above atom yields the atom `assign_6(X1,X2,X3)`. We can see in the original atom above that the first argument is the imperative program point and the second is the variable environment relating imperative and logic variables for the scope of that predicate's clause. In our case we use both versions of the residual program. Namely, we use the small version for the analyser and the verbose one to systematically relate program points to clause heads and variables in both worlds. Given the variable environment we may readily obtain the correspondence between logic variables and imperative variables, and between clause heads and program points. Interpreting the above results yields:

```
(1)  i := 2;
(2)  j := 0;
      {j}>=0, 2j+2=<i}
(3)  while (n*n > 1) do
      {j}>=0, 2j+2=<i}
(4)    if (n*n = 2) then
      {j}>=0, 2j+2=<i}
(5)      i := i+4;
      else
```

```

                {j>=0, 2j+2=<i}
(6)      i := i+2;
                {j>=0, 2j+4=<i}
(7)      j := j+1;
        endwhile

```

Another Example. Next we present another example using lists to denote arrays. This code sorts an array of size n using the bubblesort algorithm of [9]. This example was adapted from one used by Cousot and Halbwachs [2]. The analysis results appear as comments inside curly brackets along the code.

```

(1)  b := n;
(2)  while (b>=1) do
                                {n>=b, b>=1}
(3)      j := 1;
(4)      t := 0;
(5)      while (j <= b-1)
                                {n>=b, t>=0, j>=t+1, b>=j+1}
(6)          if (k[j] > k[j+1]) then
                                {n>=b, t>=0, j>=t+1, b>=j+1}
(7)              tm := k[j];
(8)              k[j] := k[j+1];
(9)              k[j+1] := tm;
(10)             t := j;
                                {n>=b, j>=1, t>=0, b>=j+1, j>=t}
(11)         j := j+1
(12)         if (t == 0) then
                                {n>=b, t=0, j>=1, b>=j, b<j+1}
(13)             b := -1;
                else
                                {n>=b, t>=0, j>=t+1, b>=j, b<j+1}
(14)             b := t;

```

These results are the same as those obtained in [2].

6 Related Work

The first practical results on imperative languages for deriving linear equality or disequality relations among the variables of a program is due to Cousot and Halbwachs [2]. Their system was implemented in Pascal. The model execution used is based on flow-charts and an approximation method based on convex polyhedra. Incidentally the analyser used on the experiments here reported uses a similar approximation method integrated with other constraint solvers [20]. Later on in [3] the author poses the possibility of deriving different static analysers parameterised by the language semantics. In a similar way [8] show how to

obtain a static analyser for a non strict functional language. Such a static analyser is derived by successive refinements of the original language specification, natural operational semantics. The possible analyses obtained by the analyser derived with this method depend on the program property sought. This program property should be provided in advance. It appears that this technique has been applied to obtain some classical compiler analyses of programs in the sense of [1]. A good source of related work on implementation/derivation of static analysers from operational semantics for different programming languages is [8]. In [21] the authors describe a technique based on the style of abstract interpretation to statically estimate the ranges of variables throughout a program. Their implementation has been realised in the context of an optimising/parallelising compiler for C. Again, this is an example of using a variant of operational semantics to describe the abstract interpreters for static analysis of imperative programs.

In [13] the author lays out the theory of abstract interpretation using two-level semantics. Two-level semantics had been previously used in [15] to describe code generation. A summary of both can be found in [14]. Using denotational definitions the semantics of Pascal-like languages is given making explicit the distinction between compile time computations and run time computations, hence the two levels of the metalanguage. For program analysis, an appropriate interpretation of the run time metalanguage aids the analysis by giving a nonstandard semantics to run time constructs. By contrast in the present work, the semantic definitions are given in a standard way, and the translation is carried out by the partial evaluator where the distinction between compile time and run time computations is accounted for.

Another sort of problem reduction for analysis is provided in [17]. The authors convert the problem of identifying dependences among program elements to a problem of identifying may-aliases. The transformation output is a program in the same language as the input program where may-aliases in the transformed program yield information directly translatable to control flow dependences between statements in the original program. In a similar way the authors claim that control flow dependences in the transformed program have a direct reading as may-aliases in the corresponding program. Presumably the their method and ours could be combined to obtain other problem reduction results.

In [19] it is shown how to use logic programs to aid the analysis of imperative programs with pointers. The formalism is shown for the case of the pointer equality problem in Pascal. During analysis a set of assertions, represented as unary clauses, is updated according to the meaning of the program statement evaluated, the update operation designated and a set of consistency rules. The update operations resemble operations in deductive databases. The semantics of the imperative program is not explicitly represented as a logic program as in our approach, but in both approaches logic programs are used to express program properties.

7 Final Remarks

We have developed a language-independent method for analysing imperative programs. The method is based on encoding the semantics of an imperative language in a logic programming language for which there are advanced tools available for program analysis and transformation. This allows us to transfer the results of research on analysis of logic programs to the analysis of imperative programs.

The emphasis of our work is to find practical and efficient techniques for achieving this aim. A key aspect is to write the semantics in a way that is amenable to analysis. We identified the one-state small-step semantics as a suitable style. The problem of relating the results of analysis of a logic program to the source code of the original imperative program was also solved. A representation of the imperative program was constructed in which program points were represented by special terms in the logic program. The partial evaluation algorithm was modified to exploit these terms, and thus produce a residual logic program whose structure mirrored that of the imperative program. Thus results of analysis of the logic programs could be related directly to the imperative code.

The correctness of our results follow from correctness of the partial evaluator and correctness of the analyser. Both correctness proofs may be done independently of the imperative language to be analysed, which we claim is one of the contributions of the present work.

Future Work

We have performed some promising experiments on a simple language, as shown in this paper, but our aim is to analyse programs in a mainstream imperative language. Currently we are well advanced in writing the operational semantics for a significant subset of Java. We aim to enhance our current analysis tools by handling non-linear arithmetic constraints, and boolean constraints. Moreover, we aim to increase the flexibility of analysis by using pre-interpretations to express properties and abstract compilation to encode them as logic programs [18].

The use of the same method to perform other analyses, such as context-sensitive or alias analyses remains to be explored.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.
2. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Conference Record of the 5th ACM Symposium on Principles of Programming Languages*, pages 84–97, Albuquerque, New Mexico, 1978.
3. Patrick Cousot. Abstract interpretation based static analysis parametrized by semantics. In *Proceedings of the Fourth International Symposium on Static Analysis, SAS'97*, pages 388–394, Paris, France, 1997. LNCS 1302, Springer Verlag.

4. J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, Denmark, 1993. ACM Press.
5. J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In *Proceedings of Meta90 Workshop on Meta Programming in Logic*. Katholieke Universiteit Leuven, Belgium, 1990.
6. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3&4):305–333, 1991.
7. J. Gallagher and D.A. de Waal. Fast and precise regular approximation of logic programs. In P. Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming (ICLP'94)*, Santa Margherita Ligure, Italy. MIT Press, 1994.
8. Valérie Gouranton and Daniel Le Métayer. Formal development of static program analysers. In *Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering*, pages 101–110, Israel, 1997.
9. Donald E. Knuth. *The Art of Computer Programming*, volume 3 of *Sorting and Searching*. Addison-Wesley Publishing Company, 1973.
10. Robert Kowalski. *Logic for Problem Solving*. North Holland, 1979.
11. Michael Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, Katholieke Universiteit Leuven, Department of Computer Science, Leuven, Belgium, May 1997.
12. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3&4):217–242, 1991.
13. Flemming Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, (69):117–242, 1989.
14. Flemming Nielson. Semantics-directed program analysis: A toolmaker's perspective. In *Third International Symposium, SAS'96*. Springer Verlag, LNCS 1145, 1996.
15. Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, (56):59–133, 1988.
16. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications*. John Wiley and Sons, 1992.
17. John L. Ross and Mooly Sagiv. Building a bridge between pointer aliases and program dependences. In *European Symposium On Programming*, Lisbon, Portugal, 1998.
18. Hüseyin Sağlam. *A Toolkit for Static Analysis of Constraint Logic Programs*. PhD thesis, Bristol University, Department of Computer Science, Bristol, U.K., March 1998.
19. Mooly Sagiv, Nissim Francez, Michael Rodeh, and Reinhard Wilhelm. A logic-based approach to program flow analysis. 1998. Submitted to Acta Informatica.
20. H. Sağlam and J. Gallagher. Constrained regular approximation of logic programs. In N. Fuchs, editor, *Logic Program Synthesis and Transformation (LOPSTR'97)*. Springer-Verlag, Lecture Notes in Computer Science, 1998. in press.
21. Clark Verbrugge, Phong Co, and Laurie Hendren. Generalized constant propagation a study of C. *Lecture Notes in Computer Science, Compiler Construction*, (1060):74–90, 1996.

Improving Control in Functional Logic Program Specialization^{*}

E. Albert¹, M. Alpuente¹, M. Falaschi², P. Julián³, and G. Vidal¹

¹ DSIC, U. Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain.
`{ealbert,alpuente,gvidal}@dsic.upv.es`

² Dip. di Mat. e Informatica, U. Udine, Via delle Scienze 206, 33100 Udine, Italy.
`falaschi@dimi.uniud.it`

³ Dep. de Informática, Ronda de Calatrava s/n, 13071 Ciudad Real, Spain.
`pjulian@inf-cr.uclm.es`

Abstract. We have recently defined a framework for Narrowing-driven Partial Evaluation (NPE) of functional logic programs. This method is as powerful as *partial deduction* of logic programs and *positive supercompilation* of functional programs. Although it is possible to treat complex terms containing primitive functions (e.g. conjunctions or equations) in the NPE framework, its basic control mechanisms do not allow for effective polygenetic specialization of these complex expressions. We introduce a sophisticated unfolding rule endowed with a dynamic narrowing strategy which permits flexible scheduling of the elements (in conjunctions) which are reduced during specialization. We also present a novel abstraction operator which extends some partitioning techniques defined in the framework of *conjunctive partial deduction*. We provide experimental results obtained from an implementation using the INDY system which demonstrate that the control refinements produce better specializations.

1 Introduction

Functional logic programming languages allow us to integrate some of the best features of the classical declarative paradigms, namely functional and logic programming. Lazy, efficient, functional computations are combined with the expressivity of logic variables, which allows for function inversion as well as logical search. The operational semantics of functional logic languages is usually based on (some form of) narrowing, which is a unification-based, parameter-passing mechanism which extends functional evaluation through goal solving capabilities as in logic programming (see [14] for a survey). In order to avoid unnecessary computations and to compute with infinite data structures, most recent work has concentrated on *lazy narrowing strategies* [11,15,23,25].

The aim of *partial evaluation* (PE) is to specialize a given program w.r.t. part of its input data (hence, also called *program specialization*). PE techniques

^{*} This work has been partially supported by CICYT TIC 95-0433-C03-03, by HCM project CONSOLE, and by Acción Integrada HA1997-0073.

have been widely applied to the optimization of functional (see [16] and references therein) and logic programs [10,18,22,27]. Unfortunately, these techniques generally cannot be easily transferred to functional logic languages, since logical variables in function calls place specific demands that have to be tackled in order to achieve effective specialization.

Narrowing-driven PE [4] (NPE) provides a general scheme for the specialization of functional logic languages. The method is formalized within the theoretical framework established in [22,24] for the PE of logic programs (also known as *partial deduction*, PD). However, a number of concepts have been generalized for dealing with features such as nested function calls, eager and lazy evaluation strategies and the standard optimization based on deterministically reducing functions. Control issues are managed by using standard techniques as in [24,28]. The NPE method of [4] distinguishes a *local* and a *global* levels of control. At the local level, (finite) narrowing trees for (nested) function calls are constructed. At the global level, the calls extracted from the leaves of the local trees are considered for the next iteration of the algorithm, after a proper abstraction (generalization) that guarantees that only a finite number of calls is specialized. A close, automatic approach is that of positive supercompilation (PS) [29], whose basic transformation operation is *driving*, a unification-based transformation mechanism which is similar to (lazy) narrowing. A different PE method for a rewriting-based, functional logic language is considered in [19].

Classical PD computes partial evaluations for separate atoms independently. Recently, [12,21] have introduced a technique for the partial deduction of conjunctions of atoms. This technique achieves a number of program optimizations such as (some form of) tupling and deforestation which are usually obtained through more expensive fold/unfold transformations, which are difficult to automate and which cannot be obtained through classical PD.

The NPE method of [4] is able to produce *polygenetic* specializations [13], i.e. it is able to extract specialized definitions which combine several function definitions of the original program. That means that NPE has the same potential for specialization as conjunctive PD or PS within the considered paradigm (a detailed comparison can be found in [5,6]). This is because the generic method of [4] may allow one to deal with equations and conjunctions during specialization by simply considering the equality and conjunction operators as *primitive* function symbols of the language. Unfortunately, the use of primitive functions may encumber the nature of the specialization problems and it often turns out that some form of *tupling* (as defined in [27] for logic programs) is required for specializing expressions which contain conjunctive calls. The NPE algorithm of [4] does not incorporate a specific treatment for such primitive symbols, which depletes many opportunities for reaching the *closedness* condition and forces the method to dramatically generalize calls, thus giving up the intended specialization (see Example 1). Inspired by the challenging results of conjunctive PD in [12], this paper extends [4,3] by formulating and experimentally testing concrete NPE control options that effectively handle primitive function symbols in lazy functional logic languages.

Some of the original contributions of our paper are as follows: i) we introduce a well-balanced dynamic unfolding rule and a novel abstraction operator that do not depend on the narrowing strategy and which highly improve the specialization of the NPE method; ii) these options allow us to tune the specialization algorithm to handle conjunctions (and other expressions containing primitive functions, such as conditionals and strict equalities) in a natural way, which provides for polygenetic specialization without any ad-hoc artifice; and iii) our method is applicable to modern functional logic languages with a lazy narrowing semantics such as Babel [25], Curry [15] and Toy [8], thus giving a specialization method which subsumes both lazy functional and conventional logic program specialization. We demonstrate the quality of these improvements by specializing some examples which were not handled well by classical NPE. The control strategies have been tested in the NPE system INDY [2].

The structure of the paper is as follows. Section 2 contains basic definitions. Section 3 extends the NPE algorithm of [3] to care for the appropriate handling of primitive function symbols. The algorithm is still generic in that no concrete control options are settled (i.e., there is no commitment to any concrete unfolding rule or abstraction operator). In Sect. 4, the concrete control options are described by formalizing some appropriate unfolding and abstraction operators. We illustrate the usefulness of our approach through some simple examples. Preliminary performance results, given in Sect. 5, show the practical importance of the proposed strategies. Finally, Sect. 6 concludes the paper. An extended version of this paper containing more details and proofs can be found in [1].

2 Preliminaries

We briefly summarize some well-known results about rewrite systems and functional logic programs [9, 14]. The definitions below are given in the homogeneous case. The extension to many-sorted signatures is straightforward [26].

Throughout this paper, \mathcal{X} denotes a countably infinite set of *variables* and \mathcal{F} denotes a set of *function symbols* (also called the *signature*), each of which has a fixed associated arity. We assume that the signature \mathcal{F} is partitioned into two sets $\mathcal{F} = \mathcal{C} \cup \mathcal{D}$ with $\mathcal{C} \cap \mathcal{D} = \emptyset$. Symbols in \mathcal{C} are called *constructors* and symbols in \mathcal{D} are called *defined functions*. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denotes the set of *terms* or *expressions* built from \mathcal{F} and \mathcal{X} . $\mathcal{T}(\mathcal{F})$ denotes the set of *ground terms*, while $\mathcal{T}(\mathcal{C}, \mathcal{X})$ denotes the set of *constructor terms*. If $t \notin \mathcal{X}$, then $\text{Head}(t)$ is the function symbol heading term t , also called the *root symbol* of t . A *pattern* is a term of the form $f(d_1, \dots, d_n)$ where $f/n \in \mathcal{D}$ and d_1, \dots, d_n are constructor terms. $\text{Var}(s)$ is the set of variables occurring in the syntactic object s .

A *substitution* is a mapping from \mathcal{X} to $\mathcal{T}(\mathcal{F}, \mathcal{X})$ s.t. its domain $\text{Dom}(\sigma) = \{x \in \mathcal{X} \mid x\sigma \neq x\}$ is finite. We frequently identify a substitution σ with the set $\{x \mapsto x\sigma \mid x \in \text{Dom}(\sigma)\}$. We denote the identity substitution by *id*. We consider the usual preorder on substitutions \leq : θ is *more general* than σ (in symbols $\theta \leq \sigma$) iff $\exists \gamma. \sigma \equiv \theta\gamma$. The *restriction* $\sigma|_V$ of a substitution σ to a set

V of variables is defined by $\sigma|_V = x\sigma$ if $x \in V$ and $\sigma|_V = x$ if $x \notin V$. We write $\sigma =_V \theta$ iff $\sigma|_V = \theta|_V$, and $\sigma \leq_V \theta$ iff $\exists \gamma. \sigma\gamma =_V \theta$.

A term t is *more general* than s (or s is an *instance* of t), in symbols $t \leq s$, if $\exists \sigma. t\sigma \equiv s$. A *unifier* of a pair of terms $\{t_1, t_2\}$ is a substitution σ such that $t_1\sigma \equiv t_2\sigma$. A unifier σ is called *most general unifier* (*mgu*) if $\sigma \leq \sigma'$ for every other unifier σ' . A *generalization* of a set of terms $\{t_1, \dots, t_n\}$ is a pair $\langle t, \{\theta_1, \dots, \theta_n\} \rangle$ such that $t\theta_i = t_i$, $i = 1, \dots, n$. A generalization $\langle t, \Theta \rangle$ is the *most specific generalization* (*msg*) if $t' \leq t$ for every other generalization $\langle t', \Theta' \rangle$.

Positions of a term t are represented by sequences of natural numbers used to address subterms of t . They are ordered by the prefix ordering: $p \leq q$ if there is w such that $p.w = q$, where $p.w$ denotes the concatenation of sequences p and w . We let Λ denote the empty sequence. $\mathcal{Pos}(t)$ and $\mathcal{FPos}(t)$ denote, respectively, the set of positions and the set of nonvariable positions of the term t . $t|_p$ is the subterm of t at position p . $t[s]_p$ is the term t with the subterm at position p replaced with s .

We find it useful to simplify our description by limiting the discussion to unconditional term rewriting systems. A *rewrite rule* is pair $l \rightarrow r$ with $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, and $\text{Var}(r) \subseteq \text{Var}(l)$. l and r are called the *left-hand side* (lhs) and *right-hand side* (rhs) of the rewrite rule, respectively. A *term rewriting system* (TRS) \mathcal{R} is a finite set of rewrite rules. A *rewrite step* is an application of a rewrite rule to a term, i.e. $t \rightarrow_{p, l \rightarrow r} s$ if there exists a position $p \in \mathcal{Pos}(t)$, a rewrite rule $l \rightarrow r$, and a substitution σ with $t|_p = l\sigma$ and $s = t[r\sigma]_p$. We say that $t|_p$ is a *redex* (reducible expression) of t . A term t is *reducible* to term s if $t \rightarrow^* s$. A term t is *irreducible* or in *normal form* if there is no term s with $t \rightarrow s$. A TRS is *terminating* if there are no infinite sequences of the form $t_1 \rightarrow t_2 \rightarrow \dots$. A TRS is called *confluent* if, whenever a term s reduces to two terms t_1 and t_2 , both t_1 and t_2 reduce to the same term. Since we do not require terminating rules, normal forms may not exist.

Functional Logic Programming

The operational semantics of functional logic programs is usually based on (some variant) of *narrowing*. Essentially, narrowing consists of computing an appropriate substitution so that when applied to the current term, it becomes reducible, and then reducing it [14]. In this section, we briefly introduce a functional logic language whose syntax and demand-driven reduction mechanism is essentially equivalent to that of (a subset of) Babel [23,25], Toy [8], and Curry [15], which has been recently proposed to become a standard in the area.

A TRS \mathcal{R} is *constructor-based* (CB) if for each rule $l \rightarrow r \in \mathcal{R}$ the lhs l is a pattern. A CB TRS \mathcal{R} is *weakly-orthogonal* if \mathcal{R} is left-linear (i.e., for each rule $l \rightarrow r \in \mathcal{R}$, the lhs l does not contain multiple occurrences of the same variable) and \mathcal{R} contains only trivial overlaps (i.e., if $l \rightarrow r$ and $l' \rightarrow r'$ are variants of distinct rules in \mathcal{R} and σ is a unifier for l and l' , then $r\sigma \equiv r'\sigma$). It is well-known that weakly-orthogonal TRS's are confluent. We henceforth consider CB weakly-orthogonal TRS's as programs. For this class of programs, a term t is a *head normal form* if t is a variable or $\text{Head}(t) \in \mathcal{C}$.

The signature \mathcal{F} is augmented with a set of primitive function symbols $\mathcal{P} = \{\approx, \wedge, \Rightarrow\}$ in order to handle complex expressions containing equations $s \approx t$, conjunctions $b_1 \wedge b_2$, and conditional (guarded) terms $b \Rightarrow t$, i.e. $\mathcal{F} = \mathcal{C} \cup \mathcal{D} \cup \mathcal{P}$. We usually treat the symbols in \mathcal{P} as infix operators. We assume that the following *predefined rules* belong to any given program:

$$\begin{array}{ll} c \approx c \rightarrow \text{true} & \% c/0 \in \mathcal{C} \\ c(x_1, \dots, x_n) \approx c(y_1, \dots, y_n) \rightarrow (x_1 \approx y_1) \wedge \dots \wedge (x_n \approx y_n) & \% c/n \in \mathcal{C} \\ \text{true} \wedge x \rightarrow x & x \wedge \text{true} \rightarrow x \quad (\text{true} \Rightarrow x) \rightarrow x \end{array}$$

These rules are weakly-orthogonal and define the validity of an equation as a *strict equality* between terms, which is common in functional languages when computations may not terminate [11,25]. Note that, although the basic computation model only supports unconditional rules, it is still adequate to support logic programs since conditional rewrite rules $l \rightarrow r \Leftarrow C$ can be encompassed by guarded unconditional rules $l \rightarrow (C \Rightarrow r)$ by using the conditional primitive ‘ \Rightarrow ’ as in Babel [25]. For reasons of simplicity, we assume the associativity of ‘ \wedge ’ and assume that ‘ \approx ’ binds more than ‘ \wedge ’ and ‘ \wedge ’ binds more than ‘ \Rightarrow ’.

We consider that programs are executed by *lazy narrowing*, which allows us to deal with nonterminating functions [23,25]. Roughly speaking, laziness means that a given expression is only narrowed at inner positions if they are *demanded* (by the pattern in the lhs of some rule) and this contributes to a later narrowing step at an outer position. Formally, given a program \mathcal{R} , we define the *one-step narrowing* relation as follows. A term s narrows to t in \mathcal{R} , in symbols $s \rightsquigarrow_{p,l \rightarrow r, \sigma} t$ (or simply $s \rightsquigarrow_{\sigma} t$), iff there exists a position $p \in \varphi(s)$, a (standardized apart) rule $l \rightarrow r \in \mathcal{R}$, and a substitution σ such that $\sigma = \text{mgu}(\{s|_p, l\})$ and $t = (s[r]_p)\sigma$. The *selection strategy* $\varphi(t)$ is responsible for computing the set of *demanded* positions of a given term t . A formal definition of this strategy in terms of an inference system is shown in [1]. Lazy narrowing is *strong complete* w.r.t. constructor substitutions in CB, weakly-orthogonal TRS’s [25, 14]. This means that the interpreter is free to disregard from $\varphi(t)$ all components of each conjunction which may occur in t except one, even if all arguments are demanded by the predefined rules of ‘ \wedge ’ (that is, completeness holds for all scheduling policies). A formal definition can be found in [3].

If $s_0 \rightsquigarrow_{\sigma_1} s_1 \rightsquigarrow_{\sigma_2} \dots \rightsquigarrow_{\sigma_n} s_n$ (in symbols, $s_0 \rightsquigarrow_{\sigma}^* s_n$, $\sigma = \sigma_1 \sigma_2 \dots \sigma_n$), we speak of a lazy narrowing *derivation* for the *goal* s_0 with (partial) *result* s_n . A lazy narrowing derivation $s \rightsquigarrow_{\sigma}^* t$ is *successful* iff $t \in \mathcal{T}(\mathcal{C} \cup \mathcal{X})$, where $\sigma|_{\text{Var}(s)}$ is the *computed answer substitution*.

3 The Generalized Specialization Algorithm

In this section, we generalize some basic concepts and techniques for the NPE of (lazy) functional logic programs (as presented in [3]). These extended notions will prove to be extremely useful for formulating new unfolding and abstraction operators which are well-suited to cope with primitive function symbols. In the original NPE framework, no distinction is made between primitive and defined function symbols during specialization. For instance, a conjunction $b_1 \wedge b_2$ is

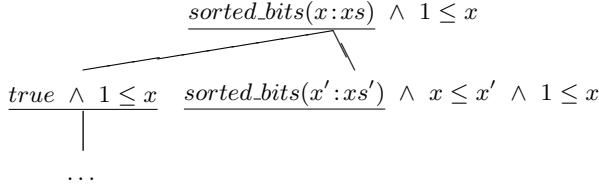


Fig. 1. Incomplete narrowing tree for $\text{sorted_bits}(x:xs) \wedge 1 \leq x$

considered as a single entity when checking whether it is covered by the set of specialized calls. This commonly implies a drastic generalization of the involved calls, which causes losing all specialization, as the following example illustrates.

Example 1. Let us consider the program excerpt:

$\text{sorted_bits}(x:[]) \rightarrow \text{true}$

$\text{sorted_bits}(x_1:x_2:xs) \rightarrow \text{sorted_bits}(x_2:xs) \wedge x_1 \leq x_2$

$0 \leq 0 \rightarrow \text{true} \quad 0 \leq 1 \rightarrow \text{true} \quad 1 \leq 1 \rightarrow \text{true}$

and the call $\text{sorted_bits}(x:xs) \wedge 1 \leq x$. The lazy narrowing tree depicted¹ in Fig. 1 is built up by using the *nonembedding* unfolding rule of [3], which expands derivations while new redexes are not “greater” (with the *homeomorphic embedding ordering*, see e.g. [4,28]) than previous, *comparable* redexes in the branch (i.e., redexes with the same outermost function symbol). From this tree, we can identify two main weaknesses of the plain NPE algorithm:

- The rightmost branch stops because the leftmost redex $\text{sorted_bits}(x':xs')$ “embeds” the previous redex $\text{sorted_bits}(x:xs)$, even if no reductions have been performed on the other elements of the conjunction.
- At the global level, since the call $\text{sorted_bits}(x':xs') \wedge x \leq x' \wedge 1 \leq x$ in the leaf of the tree embeds (but does not cover) the specialized call $\text{sorted_bits}(x:xs) \wedge 1 \leq x$ (and they are comparable), the msg $\text{sorted_bits}(x:xs) \wedge z$ is computed, which gives up the intended specialization.

The first drawback pointed out in this example motivates the definition of more sophisticated unfolding rules which are able to achieve a *balanced* evaluation of the given expression by narrowing appropriate redexes (e.g., by using some kind of dynamic scheduling strategy which takes into account the ancestors narrowed in the same branch). The second drawback suggests the definition of a more flexible abstraction operator which is able to automatically split complex terms before attempting folding or generalization. In the following, we refine the framework of [3] in order to overcome these problems.

The PE of a term is obtained by constructing a (partial) narrowing tree and then extracting the resultants associated to the root-to-leaf paths of the tree.

Definition 1 (resultant). *Let s be a term and \mathcal{R} be a program. Given a lazy narrowing derivation $s \rightsquigarrow_{\sigma}^* t$, its associated resultant is the rewrite rule $\sigma \rightarrow t$.*

¹ We assume a fixed left-to-right selection of components within conjunctions and underline the selected redex at each step.

Definition 2 (partial evaluation). Let \mathcal{R} be a program and s be a term. Let τ be a finite (possibly incomplete) narrowing tree for s in \mathcal{R} such that no goal in the tree is narrowed beyond its head normal form. Let $\{t_1, \dots, t_n\}$ be the terms in the leaves of τ . Then, the set of resultants for the narrowing sequences $\{s \rightsquigarrow_{\sigma_i}^+ t_i \mid i = 1, \dots, n\}$ is called a partial evaluation of s in \mathcal{R} .

The partial evaluation of a set of terms S in \mathcal{R} is defined as the union of the partial evaluations for the terms in S .

Intuitively, the reason for requiring that the PE of a term s do not surpass its head normal form is that, at runtime, the evaluation of a (nested) call $C[s]_p$ containing the partially evaluated term s at some position p might not demand evaluating s beyond its head normal form. Since the “contexts” $C[\]$ in which s will appear are not known at PE time, we avoid interfering with the “lazy nature” of computations in the specialized program by imposing this condition.

A recursive *closedness* condition is formalized by inductively checking that the different calls in the rules are sufficiently covered by the specialized functions.

Definition 3 (closedness). Let S be a finite set of terms. A term t is S -closed if $\text{closed}(S, t)$ holds, where the predicate *closed* is defined inductively as follows:

$$\text{closed}(S, t) \Leftrightarrow \begin{cases} \text{true} & \text{if } t \in \mathcal{X} \\ \text{closed}(S, t_1) \wedge \dots \wedge \text{closed}(S, t_n) & \text{if } t \equiv c(t_1, \dots, t_n) \\ \exists s \in S^+. s\theta = t \wedge \bigwedge_{x/t' \in \theta} \text{closed}(S, t') & \text{if } t \equiv f(t_1, \dots, t_n) \end{cases}$$

where $c \in \mathcal{C}$, $f \in (\mathcal{D} \cup \mathcal{P})$, and $S^+ = S \cup \{p(x, y) \mid p \in \mathcal{P}\}$. We say that a set of terms T is S -closed, written $\text{closed}(S, T)$, if $\text{closed}(S, t)$ holds for all $t \in T$, and we say that a program \mathcal{R} is S -closed if $\text{closed}(S, \mathcal{R}_{\text{calls}})$ holds. Here we denote by $\mathcal{R}_{\text{calls}}$ the set of terms in the rhs's of the rules in \mathcal{R} .

The novelty w.r.t. [43] is that a complex expression headed by a primitive function symbol, such as a conjunction, is proved closed w.r.t. S either by checking that it is an instance of a call in S (followed by an inductive test of the subterms), or by splitting it into two conjuncts and then trying to match with “simpler” terms in S (which happens when matching is first attempted w.r.t. one of the ‘flat’ calls $p(x, y)$ in S^+). This extension is safe since the rules which define the primitive functions are automatically added to each program.

The way in which a concrete PE is made is given by an *unfolding rule* (which decides how to stop the construction of lazy narrowing trees) and an *abstraction operator* (which ensures the finiteness of the set of specialized calls).

Definition 4 (unfolding rule [3]). An unfolding rule U is a mapping which, when given a program \mathcal{R} and a term s , returns a concrete PE for s in \mathcal{R} (a set of resultants). By $U(S, \mathcal{R})$ we denote the union of $U(s, \mathcal{R})$ for all $s \in S$.

Definition 5 (abstraction operator). Given a finite set of terms T and a set of terms S , an abstraction operator is a function which returns a finite set of terms $\text{abstract}(S, T)$ such that: i) if $s \in \text{abstract}(S, T)$, then there exists $t \in (S \cup T)$ such that $t|_p = s\theta$ for some position p and substitution θ ; ii) for all $t \in (S \cup T)$, t is closed w.r.t. the set of terms in $\text{abstract}(S, T)$.

Roughly speaking, the first condition guarantees that the abstraction operator does not introduce new function symbols, while the second condition ensures that the resulting set of terms “covers” the calls previously specialized and that closedness is preserved throughout successive abstractions.

The following basic algorithm for NPE is parameterized by the unfolding rule U and the abstraction operator $abstract$ in the style of [10].

Algorithm 1.

Input: a program \mathcal{R} and a set of terms T

Output: a set of terms S

Initialization: $i := 0$; $T_0 := T$

Repeat

1. $\mathcal{R}' := U(T_i, \mathcal{R})$;
2. $T_{i+1} := abstract(T_i, \mathcal{R}'_{calls})$;
3. $i := i + 1$;

Until $T_i = T_{i-1}$ (modulo renaming)

Return $S := T_i$

The output of the NPE algorithm, given a program \mathcal{R} , is not a PE, but a set of terms S from which the partial evaluations $U(S, \mathcal{R})$ are automatically derived. Note that, whenever the specialized call is not a pattern, lhs's of resultants are not patterns either and hence resultants are not (CB) program rules. In [3], we introduced a post-processing renaming which is useful for producing CB rules. Roughly speaking, we construct an “independent renaming” S' of S as follows: for each term s in S , we define its independent renaming $s' = f_s(x_1, \dots, x_n)$, where x_1, \dots, x_n are the distinct variables in s in the order of their first occurrence and f_s is a new fresh function symbol. Then, we fold each call t in the rules of $U(S, \mathcal{R})$ by replacing the old call t by a call to the corresponding term t' in S' (details can be found in [3]). After the algorithm terminates, the specialized program is obtained by applying this post-processing renaming to $U(S, \mathcal{R})$.

The (partial) correctness of the NPE algorithm is stated as follows.

Theorem 2. *Given a program \mathcal{R} and a term t , if Algorithm 1 terminates by computing the set of terms S , then \mathcal{R}' and t are S -closed, where $\mathcal{R}' = U(S, \mathcal{R})$.*

The correctness of the generic algorithm is stated in the following theorem, which generalizes Theorem 4.5 of [3].

Theorem 3. *Let \mathcal{R} be a program, t a term, and S a finite set of terms. Let \mathcal{R}' be a PE of \mathcal{R} w.r.t. S such that \mathcal{R}' and t are S -closed. Let S' be an independent renaming of S , and t'' (resp. \mathcal{R}'') be a renaming of t (resp. \mathcal{R}') w.r.t. S' . Then t computes in \mathcal{R} the result d with computed answer θ iff t'' computes in \mathcal{R}'' the result d with computed answer θ' and $\theta' \leq_{var(t)} \theta$.*

4 Improving Control of NPE

In Sect. 4.1 we improve control in functional logic specialization by fixing an unfolding strategy which is specifically designed for “conjunctive specialization”.

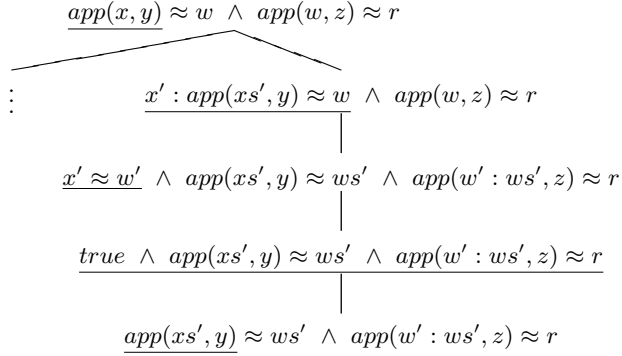


Fig. 2. Naïve local control for $app(x, y) \approx w \wedge app(w, z) \approx r$

As for global control, a specific treatment of the primitive function symbols ‘ \approx ’, ‘ \wedge ’ and ‘ \Rightarrow ’ is introduced in Sect. 4.2 which produces more effective and powerful, polygenetic specializations, as compared to classical NPE.

4.1 Local Control

The unfolding rule of [3] simply exploits the redexes selected by the lazy narrowing strategy φ (using a static selection rule which determines the next conjunct to be reduced) whenever none of them embed a previous (comparable) redex. The following example reveals that this strategy is not elaborated enough for specializing calls which may contain primitive symbols like conjunctions.

Example 2. Consider the well-known program *append*:

$$\begin{aligned}
app([], y) &\rightarrow y \\
app(x : xs, y) &\rightarrow x : app(xs, y)
\end{aligned}$$

with the input goal “ $app(x, y) \approx w \wedge app(w, z) \approx r$ ”. Using the nonembedding unfolding rule of [3], we obtain the tree depicted² in Fig. 2 (using a fixed left-to-right selection rule for conjunctions). From this local tree, no appropriate specialized definition for the initial goal can be obtained, since the leaf cannot be folded into the input call in the root of the tree and generalization is required (which causes losing all specialization, as in Example 1).

We note that the NPE method [43] succeeds with this example when the specialized call is written as a nested expression $app(app(x, y), z)$. This is because exploiting the nesting capability of the functional syntax allows us to transform the original tupling³ problem illustrated by Example 2 into a simpler, deforestation problem, which is easily solved by the original NPE method.

² We adopt the standard optimization which makes use of the built-in unification to solve strict equalities $s \approx t$ (provided that only constructor bindings are produced).

³ Here we refer to tupling of logic programs, which subsumes both deforestation and tupling of functional programs [27].

Now we introduce a dynamic lazy unfolding rule which attempts to achieve a fair evaluation of the complete input term, rather than a deeper evaluation of some given subterm. This novel concrete unfolding rule dynamically selects the positions to be reduced within a given conjunction, by exploiting some dependency information between redexes gathered along the derivation.

Definition 6 (dependent positions). Let $D \equiv (s \rightsquigarrow_{p,l \rightarrow r, \sigma} t)$ be a narrowing step. The set of dependent positions of a position q of s by D , written $q \setminus \setminus D$, is:

$$q \setminus \setminus D = \begin{cases} \{q.u \mid u \in \mathcal{FPos}(r) \wedge \text{Head}(r|_u) \notin \mathcal{C}\} & \text{if } q = p \\ \{q\} & \text{if } q \not\leq p \\ \{p.u'.v \mid r|_{u'} = x\} & \text{if } q = p.u.v \text{ and } l|_u = x \in \mathcal{X} \end{cases}$$

This notion can be naturally lifted to narrowing derivations.

The notion of dependency for terms stems directly from the corresponding notion for positions. Note that the above definition is a strict extension of the standard notion of *descendant* in functional programming. Intuitively, the descendants of the subterm $s|_q$ are computed as follows: underline the root symbol of $s|_q$ and perform the narrowing step $s \rightsquigarrow t$. Then, every subterm of t with an underlined root symbol is a descendant of $s|_q$ [17]. Intuitively, a position q' of t depends on a position q of s (by D) if q' is a descendant of q (second and third cases), or if the position q' has been introduced by the rhs of the rule applied in the reduction of the former position q and it addresses a subterm headed by a defined function symbol (first case). Note that this notion is an extension of the standard PD concept of (covering) ancestor to the functional logic framework. By abuse, we also say that the term addressed by q is an *ancestor* of the term addressed by q' in D . If s is an ancestor of t and $\text{Head}(s) = \text{Head}(t)$, we say that s is a *comparable ancestor* of t in D .

Now we formalize the way in which the dynamic selection is performed.

Definition 7 (dynamic narrowing selection strategy).

Let $D \equiv (t_0 \rightsquigarrow t_1 \rightsquigarrow \dots \rightsquigarrow t_n)$, $n \geq 0$ be a lazy narrowing derivation. We define the dynamic selection rule φ_{dynamic} as: $\varphi_{\text{dynamic}}(t_n, D) = \text{select}(t_n, \Lambda, D)$, where the auxiliary function *select* is:

$$\begin{aligned} \text{select}(t, p, D) = & \text{if } p \in \varphi(t) \\ & \text{then if } \text{dependency_clash}(t|_p, D) \text{ then } \{\perp\} \text{ else } \{p\} \\ & \text{else case } t|_p \text{ of} \\ & \quad x \in \mathcal{V}: \quad \emptyset \\ & \quad s_1 \wedge s_2: \quad \text{let } O_i = \text{select}(t, p.i, D), i \in \{1, 2\}, \text{ in} \\ & \quad \quad [\text{if } \exists i. (\perp \notin O_i \wedge O_i \neq \emptyset) \text{ then } O_i \\ & \quad \quad \text{else if } (O_1 \equiv O_2 \equiv \emptyset) \text{ then } \emptyset \text{ else } \{\perp\}] \\ & \quad \text{otherwise: } \text{let } t|_p = f(s_1, \dots, s_n) \text{ and} \\ & \quad \quad O_{\text{args}} = \bigcup_{i=1}^n \text{select}(t, p.i, D) \text{ in} \\ & \quad \quad [\text{if } \perp \in O_{\text{args}} \text{ then } \{\perp\} \text{ else } O_{\text{args}}] \end{aligned}$$

where $\text{dependency_clash}(t, D)$ is a generic Boolean function that looks at the ancestors of t in D to determine whether there is a risk of nontermination.

The following notion of *best matching terms*, which is aimed at avoiding loss of specialization due to generalization, is a proper generalization of the notion of *best matching conjunction* in [12].

Definition 8 (best matching terms). Let $S = \{s_1, \dots, s_n\}$ be a set of terms, t a term, and consider the set of terms $W = \{w_i \mid \langle w_i, \{\theta_{i1}, \theta_{i2}\} \rangle = \text{msg}(\{s_i, t\}), i = 1, \dots, n\}$. The best matching terms $BMT(S, t)$ for t in S are those terms $s_j \in S$ such that the corresponding w_j in W is a minimally general element.

The notion of BMT is used in the abstraction process at two stages: i) when selecting the more appropriate term in S which covers a new call t , and ii) when determining whether a call t headed by a primitive function symbol could be (safely) added to the current set of specialized calls or should be split.

Definition 9 (concrete abstraction operator). Let S and T be sets of terms. We define $\text{abstract}_{\triangleleft}(S, T)$ inductively as follows: $\text{abstract}_{\triangleleft}(S, T) =$

$$\begin{cases} S & \text{if } T \equiv \emptyset \text{ or } T \equiv \{t\}, t \in \mathcal{X} \\ \text{abstract}_{\triangleleft}(\dots \text{abstract}_{\triangleleft}(S, t_1), \dots, t_n) & \text{if } T \equiv \{t_1, \dots, t_n\}, n > 0 \\ \text{abstract}_{\triangleleft}(S, \{t_1, \dots, t_n\}) & \text{if } T \equiv \{t\}, t \equiv c(t_1, \dots, t_n), c \in \mathcal{C} \\ \text{abs_def}(S, T', t) & \text{if } T \equiv \{t\}, \text{Head}(t) \in \mathcal{D} \\ \text{abs_prim}(S, T', t) & \text{if } T \equiv \{t\}, \text{Head}(t) \in \mathcal{P} \end{cases}$$

where $T' = \{s \in S \mid \text{Head}(s) = \text{Head}(t) \wedge s \triangleleft t\}$. The functions abs_def and abs_prim are defined as follows:

$$\begin{aligned} \text{abs_def}(S, \emptyset, t) &= \text{abs_prim}(S, \emptyset, t) = S \cup \{t\} \\ \text{abs_def}(S, T, t) &= \text{abstract}_{\triangleleft}(S \setminus \{s\}, \{w\} \cup \text{Ran}(\theta_1) \cup \text{Ran}(\theta_2)) \\ &\quad \text{if } \langle w, \{\theta_1, \theta_2\} \rangle = \text{msg}(\{s, t\}), \text{ with } s \in BMT(T, t) \\ \text{abs_prim}(S, T, t) &= \begin{cases} \text{abs_def}(S, T, t) & \text{if } \exists s \in BMT(T, t) \text{ s.t. } \text{def}(t) = \text{def}(s) \\ \text{abstract}_{\triangleleft}(S, T, \{t_1, t_2\}) & \text{otherwise, where } t \stackrel{\wedge}{=} p(t_1, t_2) \end{cases} \end{aligned}$$

where $\text{def}(t)$ denotes a sequence with the defined function symbols of t in lexicographical order, and $\stackrel{\wedge}{=}$ is equality up to reordering of elements in a conjunction.

Essentially, the way in which the abstraction operator proceeds is simple. We distinguish the cases when the considered term i) is a variable, ii) is headed by a constructor symbol, iii) by a defined function symbol, or iv) by a primitive function symbol. The actions that the abstraction operator takes, respectively, are: i) to ignore it, ii) to recursively inspect the subterms, iii) to generalize the given term w.r.t. some of its best matching terms (recursively inspecting the $\text{msg } w$ and the subterms of θ_1, θ_2 not covered by the generalization), and iv) the same as in iii), but considering the possibility of splitting the given expression before generalizing it when $\text{def}(t) \neq \text{def}(s)$ (which essentially states that some defined function symbols would be lost due to the application of msg). The function $\text{abstract}_{\triangleleft}$ is an abstraction operator in the sense of Definition 5 [1]. The following result establishes the termination of the global specialization process

Theorem 4. Algorithm 1 terminates for the unfolding rule U_{dynamic} and the abstraction operator $\text{abstract}_{\triangleleft}$.

Our final example witnesses that $abstract_{\sqsubseteq}$ behaves well w.r.t. Example 3.

Example 4. Consider again the tree depicted in Fig. 3. By applying Algorithm 11 the following call to $abstract_{\sqsubseteq}$ is undertaken:

$$abstract_{\sqsubseteq}(\{app(x, y) \approx w \wedge app(w, z) \approx r\}, \\ \{x' \approx w' \wedge app(xs', y) \approx ws' \wedge w' \approx r' \wedge app(ws', z) \approx rs'\}).$$

Following Definition 9, by two recursive calls to abs_prim , we get:

$$\{app(x, y) \approx w \wedge app(w, z) \approx r, x' \approx w'\}.$$

By considering the independent renaming $dapp(x, y, w, z, r)$ for the specialized call $app(x, y) \approx w \wedge app(w, z) \approx r$, the method derives a (recursive) rule of the form: $dapp(x:xs, y, w:ws, z, r:rs) \rightarrow x \approx w \wedge w \approx r \wedge dapp(xs, y, ws, z, rs)$, which embodies the intended optimal specialization for this example.

5 Experiments

The refinements presented so far have been incorporated into the NPE prototype implementation system INDY (Integrated Narrowing-Driven specialization system [2]). INDY is written in SICStus Prolog v3.6 and is publicly available [2].

In order to assess the practicality of our approach, we have benchmarked the speed and specialization achieved by the extended implementation. The benchmarks used for the analysis are: **applast**, which appends an element at the end of a given list and returns the last element of the resulting list; **double_app**, see Example 2; **double_flip**, which flips a tree structure twice, then returning the original tree back; **fibonacci**, fibonacci's function; **heads&legs**, which computes the number of heads and legs of a given number of birds and cats; **match_app**, the extremely naïve string pattern matcher based on using **append**; **match_kmp**, a semi-naïve string pattern matcher; **maxlength**, which returns the maximum and the length of a list; **palindrome**, a program to check whether a given list is a palindrome; and **sorted_bits**, see Example 1. Some of the examples are typical PD benchmarks (see [20]) adapted to a functional logic syntax, while others come from the literature of functional program transformations, such as PS [29], fold/unfold transformations [7], and deforestation [30].

We have considered the following settings to test the benchmarks:

- **Evaluation strategy:** All benchmarks were executed by lazy narrowing.
- **Unfolding rule:** We have tested three different alternatives: (1) **emb_goal**: it expands derivations while new goals do not embed a previous comparable goal in the same branch; (2) **emb_redex**: the concrete unfolding rule of Sect. 4.1 which implements the *dependency_clash* test using homeomorphic embedding on comparable ancestors of selected redexes to ensure finiteness (note that it differs from **emb_goal** in that **emb_redex** implements dynamic scheduling on conjunctions and that homeomorphic embedding is checked on simple redexes rather than on whole goals); (3) **comp_redex**: the unfolding rule of Sect. 4.1 which uses the simpler definition of *dependency_clash* based on comparable ancestors of selected redexes as a whistle.
- **Abstraction operator:** Abstraction is always done as explained in Def. 9.

Table 1. Benchmark results

Benchmarks	Original		emb_goal		emb_redex		comp_redex	
	Rw	RT	Rw	Speedup	Rw	Speedup	Rw	Speedup
applast	10	90	13	1.32	28	2.20	13	1.10
double_app	8	106	39	1.63	61	1.28	15	3.12
double_flip	8	62	26	1.51	17	1.55	17	1.55
fibonacci	5	119	11	1.19	7	1.08	7	1.08
heads&legs	8	176	24	4.63	22	2.41	21	2.48
match-app	8	201	12	1.25	20	2.75	23	2.79
match-kmp	12	120	14	3.43	14	3.64	13	3.43
maxlength	14	94	51	1.17	20	1.27	18	1.25
palindrome	10	119	19	1.25	10	1.35	10	1.35
sorted_bits	8	110	16	1.15	31	2.89	10	2.68
Average	9.1	119.7	22.5	1.85	23	2.04	14.7	2.08
TMix average			1881		7441		5788	

Table 1 summarizes our benchmark results. The first two columns measure the number of rewrite rules (Rw) and the absolute runtimes (RT) for each original program. The other columns show the number of rewrite rules and the speedups achieved for the specialized programs obtained by using the three considered unfolding rules. The row at the bottom of the table (TMix) indicates the average specialization time for each considered unfolding rule. Times are expressed in milliseconds and are the average of 10 executions. Speedups were computed by running the original and specialized programs under the publicly available lazy functional logic language Toy [8]. Runtime input goals were chosen to give a reasonably long overall time. The complete code for benchmarks, the specialized calls, and the partially evaluated programs can be found in [1].

The figures in Table 1 demonstrate that the control refinements that we have incorporated into the INDY system provide satisfactory speedups on all benchmarks (which is very encouraging, given the fact that no partial input data were provided in any example, except for `match-app`, `match-kmp`, and `sorted_bits`). On the other hand, our extensions are *conservative* in the sense that there is no penalty w.r.t. the specialization achieved by the original system on non-conjunctive goals (although some specialization times are slightly higher due to the more complex processing being done). Let us note that, from the speedup results in Table 1, it can appear that there is no significant difference between the strategies `emb_redex` and `comp_redex`. However, when we also consider the specialization times (TMix) and the size of the specialized programs (Rw), we find out that `comp_redex` has a better overall behaviour. A detailed comparison between the considered unfolding strategies can be found in [1].

6 Discussion

In functional logic languages, expressions can be written by exploiting the *nesting* capability of the functional syntax, as in $app(app(x, y), z) \approx r$, but in many cases

it can be appropriate (or necessary) to decompose nested expressions as in logic programming, and write $app(x, y) \approx w \wedge app(w, z) \approx r$ (for instance, if some test such as *sorted.bits(w)* on the intermediate list *w* were necessary). The original INDY system behaves well on programs written with the “pure” functional syntax [5]. However, INDY is not able to produce good specialization on the benchmarks of Table 1 when they are written as conjunctions of subgoals. For this we could not achieve some of the standard, difficult transformations such as tupling [7, 27] within the classical NPE framework. As opposed to the classical PD framework (in which only folding on single atoms can be done), the NPE algorithm is able to perform folding on complex expressions (containing an arbitrary number of function calls). This does not suffice to achieve tupling in practice, since complex expressions are often generalized and specialization is lost. We have shown that the NPE general framework can be supplied with appropriate control options to specialize complex expressions containing primitive functions, thus providing a powerful polygenetic specialization framework with no ad-hoc setting.

To the best of our knowledge, this is the first practical framework for the specialization of modern functional logic languages with partitioning techniques and dynamic scheduling. As future research, there is room for further improvement in performance by introducing more powerful abstraction operators based on better analyses to determine the optimal way to split expressions (trying not to endanger the communication of data structures with shared variables), and by considering in practice the problem of controlling particular algebraic laws for primitive symbols.

References

1. E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. Technical Report DSIC-II/2/97, UPV, 1998. Available from URL: <http://www.dsic.upv.es/users/elp/papers.html>.
2. E. Albert, M. Alpuente, M. Falaschi, and G. Vidal. INDY User’s Manual. Technical Report, available from <http://www.dsic.upv.es/users/elp/papers.html>.
3. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of PEPM’97*, volume 32(12) of *Sigplan Notices*, pages 151–162, New York, 1997. ACM Press.
4. M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven Partial Evaluation of Functional Logic Programs. In H. Riis Nielson, editor, *Proc. of the 6th European Symp. on Programming, ESOP’96*, pages 45–61. Springer LNCS 1058, 1996.
5. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 1998. To appear.
6. M. Alpuente, M. Falaschi, and G. Vidal. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 1998. To appear.
7. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
8. R. Caballero-Roldán, F.J. López-Fraguas, and J. Sánchez-Hernández. User’s manual for Toy. Technical Report SIP-5797, UCM, Madrid (Spain), April 1997.
9. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.

10. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM'93*, pages 88–98. ACM, New York, 1993.
11. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *J. of Computer and System Sciences*, 42:363–377, 1991.
12. R. Glück, J. Jørgensen, B. Martens, and M.H. Sørensen. Controlling Conjunctive Partial Deduction of Definite Logic Programs. In *Proc. of PLILP'96*, pages 152–166. Springer LNCS 1140, 1996.
13. R. Glück and M.H. Sørensen. A Roadmap to Metacomputation by Supercompilation. In *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 137–160. Springer LNCS 1110, February 1996.
14. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
15. M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.
16. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
17. J.W. Klop and A. Middeldorp. Sequentiality in Orthogonal Term Rewriting Systems. *Journal of Symbolic Computation*, pages 161–195, 1991.
18. J. Komorowski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta-Programming in Logic*, pages 49–69. Springer LNCS 649, 1992.
19. L. Lafave and J.P. Gallagher. Partial Evaluation of Functional Logic Programs in Rewriting-based Languages. Technical Report CSTR-97-001, Department of Computer Science, University of Bristol, Bristol, England, March 1997.
20. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Tech. Rep., accessible via <http://www.cs.kuleuven.ac.be/~lpai>, 1998.
21. M. Leuschel, D. De Schreye, and A. de Waal. A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration. In M. Maher, editor, *Proc. of JICSLP'96*, pages 319–332. The MIT Press, Cambridge, MA, 1996.
22. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
23. R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In J. Penjam and M. Bruynooghe, editors, *Proc. of PLILP'93*, pages 184–200. Springer LNCS 714, 1993.
24. B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In L. Sterling, editor, *Proc. of ICLP'95*, pages 597–611. MIT Press, 1995.
25. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: the language Babel. *J. Logic Program.*, 12(3):191–224, 1992.
26. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.
27. A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
28. M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of ILPS'95*, pages 465–479. The MIT Press, 1995.
29. M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
30. P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

Directional Type Inference for Logic Programs

Witold Charatonik* and Andreas Podelski

Max-Planck-Institut für Informatik
Im Stadtwald, D-66123 Saarbrücken
{witold;podelski}@mpi-sb.mpg.de

Abstract. We follow the set-based approach to directional types proposed by Aiken and Lakshman [1]. Their type *checking* algorithm works via set constraint solving and is sound and complete for given discriminative types. We characterize directional types in model-theoretic terms. We present an algorithm for *inferring* directional types. The directional type that we derive from a logic program \mathcal{P} is uniformly at least as precise as any discriminative directional type of \mathcal{P} , i.e., any directional type out of the class for which the type *checking* algorithm of Aiken and Lakshman is sound and complete. We improve their algorithm as well as their lower bound and thereby settle the complexity (DEXPTIME-complete) of the corresponding problem.

1 Introduction

Directional types form a type system for logic programs which is based on the view of a predicate as a *directional procedure* which, when applied to a tuple of input terms, generates a tuple of output terms. There is a rich literature on types and directional types for which we can give only some entry points. Directional types occur as predicate profiles in [28], as mode dependencies in [8], and simply as types in [423]. Our use of the terminology “directional type” stems from [1].

A directional type for a program \mathcal{P} assigns input types I_p and output types O_p to each predicate p of \mathcal{P} . A program can have many directional types. For example, consider the predicate *append* defined by

$$\begin{aligned} \text{append}([], L, L). \\ \text{append}([X|Xs], Y, [X|Z]) \leftarrow \text{append}(Xs, Y, Z). \end{aligned}$$

We can give this predicate the directional type $(list, list, \top) \rightarrow (list, list, list)$, where *list* denotes the set of all lists and \top is the set of all terms, but also $(\top, \top, list) \rightarrow (list, list, list)$, as well as the least precise type $(\top, \top, \top) \rightarrow (\top, \top, \top)$. A predicate defined by a single fact $p(X)$ has a directional type $\tau \rightarrow \tau$ for all types τ .

In [1], Aiken and Lakshman present an algorithm for automatic type checking of logic programs wrt. given directional types. The algorithm runs in

* On leave from University of Wrocław, Poland. Partially supported by Polish KBN grant 8T11C02913.

NEXPTIME; they show that the problem is DEXPTIME-hard in general and PSPACE-hard for discriminative types. The algorithm works via set constraint solving; its correctness relies on a connection between the well-typedness conditions and the set constraints to which they are translated. The connection is such that the type check is sound and complete for *discriminative* types (it is still sound for general types).

Our results. In this paper, we answer two questions left open in [1]. First, we give an algorithm for *inferring* directional types. Second, we establish the DEXPTIME-completeness of the problem of directional type *checking* wrt. discriminative types.

To fix just one (the “right” one) directional type for a given logic program, we assume that the program comes with a query which, wlog., consists of one atom $main(t)$. Clearly, the choice of the type makes sense only if the input type I_{main} for the query predicate $main$ contains at least the expected set of input terms for $main$. Ideally, among all those directional types \mathcal{T} that satisfy this condition, we would like to infer the uniformly (i.e., for the input types and output types of *all* predicates) most precise one.

The uniformly most precise directional type $\mathcal{T}_{min}(\mathcal{P})$ of a program \mathcal{P} together with a specification of the query input terms does exist, as we will show. It is, however, not effectively computable in general. This is naturally the place where *abstraction* comes in. We can compute a directional type $\mathcal{T}_{sb}(\mathcal{P})$, a regular approximation of $\mathcal{T}_{min}(\mathcal{P})$ which is defined through the *set-based* abstraction à la Heintze and Jaffar [24]. There is no objective criterion to evaluate the quality of the approximation of a non-regular set by a regular one in the sense that the most precise approximation does not exist; this fact applies also to our type inference procedure. We can show, however, that $\mathcal{T}_{sb}(\mathcal{P})$ is uniformly more precise than any discriminative directional type of \mathcal{P} , i.e., any directional type out of the class for which the type *check* of Aiken and Lakshman is sound and complete.

The above comparison is interesting for intrinsic reasons and it indicates that our type inference procedure produces “good” directional types. We exploit it furthermore in order to derive a type *checking* algorithm whose complexity improves upon the one of the original algorithm in [1]. A simple refinement of the arguments given in [1] suffices to make the lower bound more precise. We thus settle the complexity (DEXPTIME-complete) of the problem of directional type checking of logic programs wrt. discriminative types.

Technically, our results are based on several basic properties of three kinds of abstraction (and their interrelation): the *set-based* abstraction (obtained by Cartesian approximation and related to our inference procedure), the *set-valued* abstraction (obtained by replacing membership constraints with set inclusions and related to the type-checking procedure for arbitrary types), and *path closure* abstraction (related to the type-checking procedure for discriminative types). These properties, that we collect in Sect. 2, are of general interest; in particular, the abstraction by path closure keeps reappearing (see, e.g., [27, 26, 31, 19, 20]). Furthermore, we establish that the directional types of a program \mathcal{P} are exactly the models of an associated logic program \mathcal{P}_{InOut} . In fact, \mathcal{P}_{InOut} is a kind of

“magic set transformation” (see, e.g., [20]) of \mathcal{P} . We obtain our results (and the soundness and completeness results in [1]) by combining the model-theoretic characterization of directional types with the properties of abstractions established in Sect. 2. In fact, by having factored out general properties of abstractions from the aspects proper to directional types, we have maybe given a new view of the results in [1].

Other related work. Rouzaud and Nguyen-Phong [28] describe a type system where types are sets of non-ground terms and express directionality, but these sets must be tuple-distributive. Our types need not be tuple-distributive. In [14], Codish and Demoen infer type dependencies for logic programs. Their techniques (abstract compilation) are quite different from ours and the derived dependencies express all possible input-output relationships. Probably the work of Heintze and Jaffar is the one that is most closely related to ours. This is not only due to the fact that we use set-based analysis [24] to approximate the type program. Some of their papers [23,25] contain examples where they compute for each predicate p a pair of sets $Call_p$ and Ret_p , which can be viewed as (ground) directional type $Call_p \rightarrow Ret_p$. We are not aware, however, of a general, formal treatment of directional types inference in their work. Also the work of Gallagher and de Waal [20,21] is closely related to ours. They use a kind of “magic set transformation” to compute query and answer predicates for each predicate in the given program, which is essentially the same as our type program. Then they approximate the success set of the new predicates with ground, regular and tuple-distributive sets, which they do not call types. Boye in [5] (see also [6,7]) presents a procedure that infers directional types for logic programs. The procedure is not fully automatic (sometimes it requires an interaction with the user), it requires the set of possible types to be finite, and no complexity analysis is given. In our approach, any regular set of terms is an admissible type; our procedure is fully automatic in the presence of a query for the program (or lower bounds for input types) and it runs in single-exponential time. We refer to [1] for comparison with still other type systems. Most of those interpret types as sets of ground terms, while we interpret types as sets of non-ground terms. Most type systems do not express the directionality of predicates.

2 Abstractions

In this section we discuss three kinds of abstraction of a given program. The motivation for this discussion is the following. In the next section we will define a type program for a given program \mathcal{P} . The models of this program are directional types of \mathcal{P} . The least model of the *set-based* abstraction of the type program correspond to the type that we infer. The models of the *set-valued* abstraction correspond to the directional types that pass the type-check from [1]. The models of the *path-closure* abstraction correspond to discriminative directional types of \mathcal{P} . Later we will use the relations between these abstractions to compare different directional types of a given program.

Preliminaries. We follow the notation and terminology of [1] unless specified otherwise. For example, we use the symbols p, q, p_0, \dots for predicates (instead of f, g, f_0, \dots as in [1]) and write $p(t)$ for predicate atoms (instead of $f\ t$). Wlog., all predicates are unary. Terms t are of the form x or $f(t_1, \dots, t_n)$. We may write $t[x_1, \dots, x_m]$ for t if x_1, \dots, x_m are the variable occurrences in t (we distinguish between multiple occurrences of the same variable), and $t[t_1, \dots, t_m]$ for the term obtained from t by substituting t_j for the occurrence x_j . A logic program \mathcal{P} is a set of Horn clauses, i.e., implications of the form $p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n)$. A program comes with a query (“the main loop”) which, wlog., is specified by one atom $\text{main}(t)$. The interpretation of programs, which is defined as usual, may be viewed as a mapping from predicate symbols to sets of trees. Programs may be viewed as formulas whose free variables are set-valued (and are referred to via predicate symbols), hence as a large class of *set constraints*.

We will not use (positive) set expressions and set constraints as in [1] but, instead, logic programs, for specifying sets of trees as well as for abstraction. Positive set expressions, which denote sets of trees, can be readily translated into equivalent *alternating* tree automata, which again form the special case of logic programs whose clauses are all of the form

$$p(f(x_1, \dots, x_n)) \leftarrow p_{11}(x_1), \dots, p_{1m_1}(x_1), \dots, p_{n1}(x_n), \dots, p_{nm_n}(x_n)$$

where x_1, \dots, x_n are pairwise different. A *non-deterministic* tree automaton is the special case where $m_1 = \dots = m_n = 1$, i.e., a logic program whose clauses are all of the form $p(f(x_1, \dots, x_n)) \leftarrow p_1(x_1), \dots, p_n(x_n)$. A set of trees is *regular* if it can be denoted by a predicate p in the least model of a non-deterministic tree automaton.

A *uniform program* [19] consists of Horn clauses in one of the following two forms. (In a *linear* term, each variable occurs at most once.)

- $p(t) \leftarrow p_1(x_1), \dots, p_k(x_m)$ where the term t is linear.
- $q(x) \leftarrow p_1(t_1), \dots, p_m(t_m)$ where t_1, \dots, t_m are any terms over $\Sigma(\text{Var})$.

A uniform program can be transformed (in single-exponential time) into an equivalent non-deterministic tree automaton [19,18,10].

Set-based abstraction. We use set-based analysis in the sense of [24] but in the formulation using logic programs as in [19,18,10] (instead of set constraints as in [24] and [1]).

Definition 1 ($\mathcal{P}^\#$, the *set-based abstraction* of \mathcal{P}). *The uniform program $\mathcal{P}^\#$ is obtained from a program \mathcal{P} by translating every clause $p(t) \leftarrow \text{body}$, whose head term t contains the n variables x_1, \dots, x_n , into the $(n+1)$ clauses*

$$\begin{aligned} p(\tilde{t}) &\leftarrow p_1(x_1^1), \dots, p_1(x_1^{m_1}), \dots, p_n(x_n^1), \dots, p_n(x_n^{m_n}) \\ p_i(x_i) &\leftarrow \text{body} \quad (\text{for } i = 1, \dots, n) \end{aligned}$$

where \tilde{t} is obtained from t by replacing the m_i different occurrences of variables x_i by different renamings $x_i^1, \dots, x_i^{m_i}$, and p_1, \dots, p_n are new predicate names.

The least model of the program $\mathcal{P}^\#$ expresses the *set-based* abstraction of \mathcal{P} , which is defined in [24] as the least fixpoint of the operator $\tau_{\mathcal{P}}$. The operator $\tau_{\mathcal{P}}$ is defined via set-based substitutions in [24]; it can also be defined by (for a subset M of the Herbrand base)

$$\begin{aligned} \tau_{\mathcal{P}}(M) = \{ & p_0(t_0[x_1\theta_1, \dots, x_m\theta_m]) \mid p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n) \in \mathcal{P}, \\ & \theta_1, \dots, \theta_m \text{ ground substitutions,} \\ & p_1(t_1\theta_1), \dots, p_n(t_n\theta_1) \in M \\ & \vdots \\ & p_1(t_1\theta_m), \dots, p_n(t_n\theta_m) \in M \} \end{aligned}$$

where x_1, \dots, x_m are the variable occurrences in t_0 (we distinguish between multiple occurrences of the same variable). As noted in [19], the logical consequence operator associated with $\mathcal{P}^\#$ is equal to the set-based consequence operator; i.e., $T_{\mathcal{P}^\#} = \tau_{\mathcal{P}}$. We recall that the logical consequence operator associated with the program \mathcal{P} is defined by

$$\begin{aligned} T_{\mathcal{P}}(M) = \{ & p_0(t_0[x_1\theta, \dots, x_m\theta]) \mid p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n) \in \mathcal{P}, \\ & \theta \text{ ground substitution,} \\ & p_1(t_1\theta), \dots, p_n(t_n\theta) \in M \}. \end{aligned}$$

The set-based abstraction can be formalized in the abstract interpretation framework [16] by the application of the *Cartesian approximation* \mathcal{C} to the semantics-defining fixpoint operator $T_{\mathcal{P}}$; i.e., $T_{\mathcal{P}^\#} = \mathcal{C}(T_{\mathcal{P}})$ (see [17]; roughly, \mathcal{C} maps a set of tuples to the smallest Cartesian product containing it). Thus, we have

$$T_{\mathcal{P}^\#} = \tau_{\mathcal{P}} = \mathcal{C}(T_{\mathcal{P}}).$$

We note the following fact, keeping symmetry with Remarks 2 and 3 on the two other abstractions of $T_{\mathcal{P}}$ that we will introduce. Given two set-valued functions F and F' , we write $F \leq F'$ if F is smaller than F' wrt. to pointwise subset inclusion, i.e., $F(x) \subseteq F'(x)$ for all x .

Remark 1 (Set-based approximation). The direct-consequence operator associated with $\mathcal{P}^\#$ approximates the one associated with \mathcal{P} ; i.e.,

$$T_{\mathcal{P}} \leq T_{\mathcal{P}^\#}.$$

Proof. Obvious by definition. □

The following statement will be used for the soundness of our type inference algorithm (Theorem 3). Its converse does, of course, not hold in general (the least models of \mathcal{P} may be strictly smaller than the least models of $\mathcal{P}^\#$).

Proposition 1. *Each model \mathcal{M} of the set-based abstraction $\mathcal{P}^\#$ of a program \mathcal{P} is also a model of \mathcal{P} .*

Proof. A ground instance of a clause of \mathcal{P} is also a ground instance of the corresponding clause of $\mathcal{P}^\#$. \square

Set-valued abstraction. The second abstraction that we consider is also defined via a program transformation: an atom $p(t)$ is simply replaced by an inclusion $t \subseteq p$. The transformed program is interpreted over the domain of *sets* of trees; i.e., the valuations are mappings $\theta : \text{Var} \rightarrow 2^T$. These mappings are extended canonically from variables x to terms t ; i.e., $t\theta$ is a set of trees. We repeat that an interpretation \mathcal{M} , i.e., a subset of the Herbrand base, maps predicates p to sets of trees $p^\mathcal{M} = \{t \in T_\Sigma \mid p(t) \in \mathcal{M}\}$. The inclusion $t \subseteq p$ holds in \mathcal{M} under the valuation θ if $t\theta$ is a subset of $p^\mathcal{M}$.

Definition 2 (\mathcal{P}^\subseteq , the set-valued abstraction of \mathcal{P}). *Given a program \mathcal{P} , its set-valued program abstraction is a program \mathcal{P}^\subseteq that is interpreted over sets of trees (instead of trees). It is obtained by replacing membership with subset inclusion; i.e., it contains, for each clause $p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n)$ in \mathcal{P} , the implication*

$$t_0 \subseteq p_0 \leftarrow t_1 \subseteq p_1, \dots, t_n \subseteq p_n. \quad (1)$$

The models of the program \mathcal{P}^\subseteq are, as one expects, interpretations \mathcal{M} (subsets of the Herbrand base) such that all implications all valid in \mathcal{M} . An implications of the form (1) is valid in \mathcal{M} if for all valuations $\theta : \text{Var} \rightarrow 2^T$, if $t_i\theta$ is a subset of $p_i^\mathcal{M}$ for $i = 1, \dots, n$ then also for $i = 0$.

The models of \mathcal{P}^\subseteq are the fixpoints of $T_{\mathcal{P}^\subseteq}$, the direct consequence operator associated with \mathcal{P}^\subseteq , which is defined in a way analogous to $T_{\mathcal{P}}$ (using set-valued substitutions instead of tree-valued substitutions). Hence, we will be able to use the following remark when we compare models of $\mathcal{P}^\#$ with models of \mathcal{P}^\subseteq (Proposition 4).

Remark 2 (Set-valued approximation). The direct-consequence operator associated with \mathcal{P}^\subseteq approximates the one associated with \mathcal{P} ; i.e.,

$$T_{\mathcal{P}} \leq T_{\mathcal{P}^\subseteq}.$$

Proof. If, for some subset M of the Herbrand base, $T_{\mathcal{P}}(M)$ contains the ground atom $p_0(t_0)$ because M contains the ground atoms $p_1(t_1), \dots, p_n(t_n)$ and $p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n)$ is a ground instance of some clause of \mathcal{P} , then the singleton $\{t_i\}$ is a subset of the denotation of p_i under M (for $i = 1, \dots, n$) and, hence, the singleton $\{t_0\}$ is a subset of the denotation of p_0 under $T_{\mathcal{P}^\subseteq}(M)$. \square

The next statement underlies the *soundness* of the type checking procedure of [11], which essentially checks if a given interpretation is a model of the program $\mathcal{P}_{InOut}^\subseteq$ defined in the next section (cf. Theorem 9 in [11]). It says that being a model wrt. \mathcal{P}^\subseteq is a sufficient condition for being a model of the program \mathcal{P} . (The model property of a regular set wrt. \mathcal{P}^\subseteq is nothing else than an entailment relation between set constraints. The entailment is equivalent to satisfiability of negative constraints and can be tested in NEXPTIME [12]).

Proposition 2. *Each model \mathcal{M} of the set-valued abstraction \mathcal{P}^\subseteq of a program \mathcal{P} is also a model of \mathcal{P} .*

Proof. If $c \equiv p(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n)$ is a ground instance of a clause of \mathcal{P} , then $\{t_0\} \subseteq p \leftarrow \{t_1\} \subseteq p_1, \dots, \{t_n\} \subseteq p_n$ is a ground instance of the corresponding implication of \mathcal{P}^\subseteq (which holds in \mathcal{M} if \mathcal{M} is a model of \mathcal{P}^\subseteq , and, thus, c also holds). \square

The converse of the statement above does not hold in general. Take, for example, the program \mathcal{P} defined by the clause $p(f(x, x)) \leftarrow q(f(x, x))$ and the four facts $q(f(a, a)), q(f(a, b)), q(f(b, a)), q(f(b, b))$. Then \mathcal{P}^\subseteq consists of the implication $f(x, x) \subseteq p \leftarrow f(x, x) \subseteq q$ and the four inclusions $f(a, a) \subseteq q, f(a, b) \subseteq q, f(b, a) \subseteq q, f(b, b) \subseteq q$. Then $\mathcal{M} = \{p(f(a, a)), p(f(b, b)), q(f(a, a)), q(f(a, b)), q(f(b, a)), q(f(b, b))\}$ is a model of \mathcal{P} but \mathcal{M} is not a model of \mathcal{P}^\subseteq . (This example transfers, in the essence, Example 1 in [1] from the setting of directional types to a general setting.) The converse of the statement in Proposition 2 does, however, hold in the special case of path closed models (Proposition 3) which we will introduce below.

The least fixpoint of $T_{\mathcal{P}^\subseteq}$ is in general not regular. To see this note that it is equal to the least fixpoint of $T_{\mathcal{P}}$ if, for example, \mathcal{P} is the length program. This example also shows that the least model of $\mathcal{P}^\#$ is in general not contained in *every* model of \mathcal{P}^\subseteq . This is the case, however, in the special case where the model of \mathcal{P}^\subseteq is path closed (Proposition 4).

Path closed models. The following notion of a path-closed set originates from [22]. It is equivalent to other notions occurring in the literature: tuple-distributive [27, 28], discriminative [1], or deterministic.

Definition 3 (Path-closed). *A [regular] set of ground terms is called path-closed if it can be defined by a [finite] deterministic top-down tree automaton.*

A deterministic finite tree automaton translates to a logic program which does not contain two different clauses with the same head (modulo variable renaming), e.g., $p(f(x_1, \dots, x_n)) \leftarrow p_1(x_1), \dots, p_n(x_n)$ and $p(f(x_1, \dots, x_n)) \leftarrow p'_1(x_1), \dots, p'_n(x_n)$. A discriminative set expression as defined in [1] translates to a deterministic finite tree automaton, and vice versa. That is, discriminative set expressions denote exactly path-closed regular sets. It is argued in [1] that discriminative set expressions are quite expressive and are used to express commonly used data structures. Note that lists, for example, can be defined by the program with the two clauses $list(cons(x, y)) \leftarrow list(y)$ and $list(nil)$.

The following fact is *the* fundamental property of path closed sets in the context of set constraints (see also Theorem 12 and Lemma 14 in [1]). It will be directly used in Proposition 3. For comparison, take the constraint $f(x, y) \subseteq f(a, a) \cup f(b, b)$; here, $\{f(a, a), f(b, b)\}$ is a set that is *not* path closed, and the union of the (set-valued) solutions $\theta_1 : x, y \mapsto \{a\}$ and $\theta_2 : x, y \mapsto \{b\}$ is *not* a solution. Also, take the constraint $f(x, y) \subseteq \emptyset$; here, the union of the solutions over possibly *empty* sets $\theta_1 : x \mapsto \{a\}, y \mapsto \emptyset$ and $\theta_2 : x \mapsto \emptyset, y \mapsto \{a\}$ is *not* a solution.

Lemma 1. *Solutions of conjunctions of inclusions $t \subseteq e$ between terms t interpreted over nonempty sets and expressions e denoting path closed sets of trees are closed under union; i.e., if S is a set of solutions, then Θ defined by $\Theta(x) = \bigcup \{\theta(x) \mid \theta \in S\}$ is again a solution.*

Proof. The statement follows from the fact (shown, e.g., in [13]) that in the interpretation over nonempty sets, if the upper bounds e denote path closed sets then inclusions of the form $f(x_1, \dots, x_n) \subseteq e$ are equivalent to the conjunction

$$x_1 \subseteq f_{(1)}^{-1}(e) \wedge \dots \wedge x_n \subseteq f_{(n)}^{-1}(e)$$

where $f_{(i)}^{-1}(e)$ denotes the set $\{t_i \mid f(t_1, \dots, t_n) \in e\}$. □

The following statement underlies the *completeness* of the type checking procedure of [1] for discriminative directional types (see Theorem 12, Lemma 14 and Theorem 15 in [1]).

Proposition 3. *Each path closed model \mathcal{M} of a program \mathcal{P} is also a model of \mathcal{P}^\subseteq , its set-valued abstraction.*

Proof. Assume that the clause $p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n)$ is valid in \mathcal{M} (i.e., it holds under all ground substitutions $\sigma : \text{Var} \mapsto T_\Sigma$, under the interpretation of the predicates p_0, p_1, \dots, p_n by \mathcal{M}), and that θ is a substitution mapping variables to nonempty sets such that

$$t_1\theta \subseteq p_1, \dots, t_n\theta \subseteq p_n$$

holds in \mathcal{M} . We have to show that also $t_0\theta \subseteq p_0$ holds in \mathcal{M} . The assumption yields that, for every ground substitution $\sigma : \text{Var} \mapsto T_\Sigma$ such that $\sigma(x) \in \theta(x)$ for all $x \in \text{Var}$,

$$t_1\sigma \in p_1, \dots, t_n\sigma \in p_n$$

holds in \mathcal{M} . Thus, also $t_0\sigma \in p_0$ holds in \mathcal{M} . Since we have that

- $t_0\sigma \in p_0$ is equivalent to $t_0\bar{\sigma} \subseteq p_0$ where $\bar{\sigma}$ is the set substitution defined by $\bar{\sigma}(x) = \{\sigma(x)\}$,
- θ is the union of all $\bar{\sigma}$ such that $\sigma(x) \in \theta(x)$ for all $x \in \text{Var}$,
- solutions of $t_0 \subseteq p_0$, where p_0 is interpreted by \mathcal{M} as a path closed set, are closed under union (Lemma [1]),

the inclusion $t_0 \subseteq p_0$ also holds in \mathcal{M} under the substitution θ . □

Path closure abstraction. The path closure PC of a set M of trees is the smallest path closed set containing M . We consider a third abstraction of the operator $T_{\mathcal{P}}$ by composing the path closure PC with $T_{\mathcal{P}}$. We note that we do not know whether the least fixpoint of the operator $PC \circ T_{\mathcal{P}}$ is always a regular set. The following comparison of two of the three abstractions that we have defined so far will be used in the proof of Proposition [4].

Remark 3 (Set-based approximation and path closure). The path closure abstraction of the direct-consequence operator of \mathcal{P} approximates also its set-based abstraction; i.e.,

$$T_{\mathcal{P}^\#} \leq (PC \circ T_{\mathcal{P}}).$$

Proof. We use the equality $T_{\mathcal{P}^\#} = \tau_{\mathcal{P}}$. If $p_0(t_0[x_1\theta_1, \dots, x_m\theta_m]) \in \tau_{\mathcal{P}}(M)$ because $p_1(t_1\theta_j), \dots, p_n(t_n\theta_j) \in M$, then $p_0(t_0\theta_j) \in T_{\mathcal{P}}(M)$, for $j = 1, \dots, m$. But then we have $p_0(t_0[x_1\theta_1, \dots, x_m\theta_m]) \in (PC \circ T_{\mathcal{P}})(M)$. \square

We will use the following statement later in order to compare the directional types obtained by our type inference procedure with the subclass of discriminative directional types for which the type check in [1] is sound and complete. (Note that the path closure of a model of a program is in general not itself a model, and that the least model of $\mathcal{P}^\#$ is in general not contained in the path closure of \mathcal{P}).

Proposition 4. *The least model of $\mathcal{P}^\#$, the set-based abstraction of a program \mathcal{P} , is contained in every path closed model of \mathcal{P}^\subseteq , the set-valued abstraction of a program \mathcal{P} .*

Proof. By Remark [3] $T_{\mathcal{P}^\#} \subseteq (PC \circ T_{\mathcal{P}})$ and thus, by Remark [2], $T_{\mathcal{P}^\#} \subseteq (PC \circ T_{\mathcal{P}^\subseteq})$. If \mathcal{M} is a path-closed model of \mathcal{P}^\subseteq , then it is also a fixpoint of $PC \circ T_{\mathcal{P}}$, and hence it contains the least fixpoint of $T_{\mathcal{P}^\#}$, i.e., the least model of $\mathcal{P}^\#$. \square

3 Directional Types and Type Programs

A *type* T is a set of terms t closed under substitution [2]. A *ground type* is a set of ground terms (i.e., trees), and thus a special case of a type. A term t has type T , in symbols $t:T$, if $t \in T$. A *type judgment* is an implication $t_1:T_1 \wedge \dots \wedge t_n:T_n \rightarrow t_0:T_0$ that holds under all term substitutions $\theta: \text{Var} \rightarrow T_\Sigma(\text{Var})$.

Definition 4 (Directional type of a program [8,1]). A directional type of a program \mathcal{P} is a family $\mathcal{T} = (I_p \rightarrow O_p)_{p \in \text{Pred}}$ assigning to each predicate p of \mathcal{P} an input type I_p and an output type O_p such that, for each clause $p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n)$ of \mathcal{P} , the following type judgments hold.

$$\begin{aligned} t_0:I_{p_0} &\rightarrow t_1:I_{p_1} \\ t_0:I_{p_0} \wedge t_1:O_{p_1} &\rightarrow t_2:I_{p_2} \\ &\vdots \\ t_0:I_{p_0} \wedge t_1:O_{p_1} \wedge \dots \wedge t_{n-1}:O_{p_{n-1}} &\rightarrow t_n:I_{p_n} \\ t_0:I_{p_0} \wedge t_1:O_{p_1} \wedge \dots \wedge t_n:O_{p_n} &\rightarrow t_0:O_{p_0} \end{aligned}$$

We then also say that \mathcal{P} is well-typed wrt. \mathcal{T} . A program together with its query $\text{main}(t)$ is well-typed wrt. \mathcal{T} if furthermore the query argument t is well-typed wrt. the input type for main (i.e., the type judgment $t:I_{\text{main}}$ holds).

Definition 5 (Ordering on directional types). We define that $\mathcal{T} = (I_p \rightarrow O_p)_{p \in \text{Pred}}$ is uniformly more precise than $\mathcal{T}' = (I'_p \rightarrow O'_p)_{p \in \text{Pred}}$ if $I_p \subseteq I'_p$ and $O_p \subseteq O'_p$ for all predicates p .

The least precise directional type for which any program (possibly together with a query) is well-typed is $\mathcal{T}_\top = (\top \rightarrow \top)_{p \in \text{Pred}}$ assigning the set of all terms to each input and output type. In the absence of a query, the most precise one is $\mathcal{T}_\perp = (\perp \rightarrow \perp)_{p \in \text{Pred}}$ assigning the empty set to each input and output type. This changes if, for example, a query of the form *main* is present (see Sect. 4).

Definition 6 ($\text{Sat}(T)$, the type of terms satisfying T [1]). Given the ground type T , the set $\text{Sat}(T)$ of terms satisfying T is the type

$$\text{Sat}(T) = \{t \in T_\Sigma(\text{Var}) \mid \theta(t) \in T \text{ for all ground substitutions } \theta : \text{Var} \rightarrow T_\Sigma\}.$$

Definition 7 (Discriminative type). A directional type is called discriminative if it is of the form $(\text{Sat}(I_p) \rightarrow \text{Sat}(O_p))_{p \in \text{Pred}}$, where the sets I_p, O_p are path-closed.

Remark 4. The clause $p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n)$ is valid in some model \mathcal{M} if and only if the type judgment

$$t_0 : \text{Sat}(p_0) \leftarrow t_1 : \text{Sat}(p_1) \wedge \dots \wedge t_n : \text{Sat}(p_n)$$

holds in \mathcal{M} (i.e., under the interpretation of p_0, p_1, \dots, p_n by \mathcal{M}).

Proof. Membership of the application of substitutions to terms in sets of the form $\text{Sat}(E)$ is defined by the application of ground substitutions to the terms in E . \square

A directional type of the form $\mathcal{T} = (\text{Sat}(I_p) \rightarrow \text{Sat}(O_p))_{p \in \text{Pred}}$, for ground types $I_p, O_p \subseteq T_\Sigma$, satisfies a type judgment if and only if the corresponding directional ground type $\mathcal{T}_g = (I_p \rightarrow O_p)_{p \in \text{Pred}}$ does.

We will next transform the well-typedness condition in Definition 4 into a logic program by replacing $t : I_p$ with the atom $p^{In}(t)$ and $t : O_p$ with $p^{Out}(t)$.

Definition 8 (\mathcal{P}_{InOut} , the type program for \mathcal{P}). Given a program \mathcal{P} , the corresponding type program \mathcal{P}_{InOut} defines an in-predicate p^{In} and an out-predicate p^{Out} for each predicate p of \mathcal{P} . Namely, for every clause $p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n)$ in \mathcal{P} , \mathcal{P}_{InOut} contains the n clauses defining in-predicates corresponding to each atom in the body of the clause,

$$\begin{aligned} p_1^{In}(t_1) &\leftarrow p_0^{In}(t_0) \\ p_2^{In}(t_2) &\leftarrow p_0^{In}(t_0), p_1^{Out}(t_1) \\ &\vdots \\ p_n^{In}(t_n) &\leftarrow p_0^{In}(t_0), p_1^{Out}(t_1), \dots, p_{n-1}^{Out}(t_{n-1}) \end{aligned}$$

and the clause defining the out-predicate corresponding to the head of the clause,

$$p_0^{Out}(t_0) \leftarrow p_0^{In}(t_0), p_1^{Out}(t_1), \dots, p_n^{Out}(t_n).$$

If the program \mathcal{P} comes together with a query $main(t)$, we add the clause $main^{In}(t) \leftarrow true$ to \mathcal{P}_{InOut} . The next statement extends naturally to a characterization of the well-typedness of a program together with a query.

Theorem 1 (Types and models of type programs). *The program \mathcal{P} is well-typed wrt. the directional type*

$$\mathcal{T} = (Sat(I_p) \rightarrow Sat(O_p))_{p \in Pred}$$

(with ground types I_p, O_p) if and only if the subset of the Herbrand base corresponding to \mathcal{T} ,

$$\mathcal{M}_{\mathcal{T}} = \{p^{In}(t) \mid t \in I_p\} \cup \{p^{Out}(t) \mid t \in O_p\},$$

is a model of the type program \mathcal{P}_{InOut} .

Proof. The validity of the well-typing conditions under *ground* substitutions is exactly the logical validity of the clauses of \mathcal{P}_{InOut} in the model $\mathcal{M}_{\mathcal{T}}$. The statement then follows by Remark 4. \square

We next define two abstractions of type programs. We will use $\mathcal{P}_{InOut}^{\#}$ for type inference (Sect. 4) and $\mathcal{P}_{InOut}^{\subseteq}$ for type checking (Sect. 5). Given a directional type \mathcal{T} , the interpretation of $\mathcal{P}_{InOut}^{\subseteq}$ by the corresponding subset $\mathcal{M}_{\mathcal{T}}$ is the *set constraint condition* which is used in [1] to replace the well-typedness condition in Definition 4.

Definition 9 (Abstractions of type programs). *The set-based type program $\mathcal{P}_{InOut}^{\#}$ is the set-based abstraction of \mathcal{P}_{InOut} , and the set-valued type program $\mathcal{P}_{InOut}^{\subseteq}$ is the set-valued abstraction of \mathcal{P}_{InOut} ; i.e.,*

$$\begin{aligned}\mathcal{P}_{InOut}^{\#} &= (\mathcal{P}_{InOut})^{\#}, \\ \mathcal{P}_{InOut}^{\subseteq} &= (\mathcal{P}_{InOut})^{\subseteq}.\end{aligned}$$

The following direct consequence of Theorem 1 and Propositions 2 and 3 restates the soundness and the conditional completeness of the type check in [1].

Theorem 2 ([Discriminative] types and models of set-valued type programs). *A directional type of the form $\mathcal{T} = (Sat(I_p) \rightarrow Sat(O_p))_{p \in Pred}$ is a directional type of the program \mathcal{P} if the corresponding subset $\mathcal{M}_{\mathcal{T}}$ of the Herbrand base is a model of $\mathcal{P}_{InOut}^{\subseteq}$, the set-valued abstraction of the type program of \mathcal{P} . For discriminative directional types \mathcal{T} , the converse also holds.*

Proof. The first part follows from Proposition 2 together with Theorem 1, the second from Proposition 3 together with Theorem 1. \square

4 Directional Type Inference

We consider three different scenarios in which we may want to infer directional types from a program.

(1) The program comes together with a query consisting of one atom *main* without arguments (and there is a clause $main \leftarrow p_1(t_1), \dots, p_n(t_n)$ calling the actual query). In this case, we are interested in inferring the most precise directional type \mathcal{T} such that \mathcal{P} together with the query *main* is well-typed.

According to the model-theoretic characterization of the well-typedness of a program together with a query (Theorem [11](#)), we must add the clause $main^{In} \leftarrow true$ to \mathcal{P}_{InOut} . This means that $\mathcal{T}_\perp = (\perp \rightarrow \perp)_{p \in \text{Pred}}$ is generally not a directional type of a program together with the query *main* and, hence, it is nontrivial to infer a precise one.

(2) The program comes together with a query consisting of one atom *main*(*t*) and a lower bound M_{main} for the input type of *main* is given (the user hereby encodes which input terms to the query predicates are expected). In this case, we are interested in inferring the most precise directional type $\mathcal{T} = (I_p \rightarrow O_p)_{p \in \text{Pred}}$ for \mathcal{P} such that the input type for *main* lies above the lower bound for *main*, i.e., such that $M_{main} \subseteq I_{main}$.

For example, take the program defining the predicate *reverse*(*x*, *y*)

$$\begin{aligned} &reverse([], []). \\ &reverse([X|Xs], Y) \leftarrow reverse(Xs, Z), append(Z, [X], Y). \end{aligned}$$

together with the definition of *append* and with the query *reverse*(*x*, *y*). If the expected input terms are lists (for *x*) and non-instantiated variables (for *y*), i.e., the lower bound specified is $M_{rev} = (list, \top)$, then the type inferred by our algorithm for *reverse* is $(list, \top) \rightarrow (list, list)$ and the type inferred by our algorithm for *append* is $(list, [\top], \top) \rightarrow (list, [\top], list)$, where $[\top]$ is the type of all singleton lists.

(3) Lower bounds M_p for the input types I_p of *all* predicates *p* of \mathcal{P} are given. This may be done explicitly for some *p* and implicitly, with $M_p = \emptyset$, for the others. In this case, we are interested in inferring the most precise directional type $\mathcal{T} = (I_p \rightarrow O_p)_{p \in \text{Pred}}$ for \mathcal{P} such that the input types for *p* lie above the given lower bounds, i.e., such that $M_p \subseteq I_p$ for all *p*.

In a setting with program modules, for example, the lower bounds that are explicitly specified may be the query goals of exported predicates.

The scenario (3) resembles the one imagined by Aiken and Lakshman in the conclusion of [11](#). Note that in our setting, however, the sets M_p need not already be input types. For example, if the inputs for *x* and *y* in *append*(*x*, *y*, *z*) are expected to be lists of even length only, i.e., the lower bound for I_{app} is given by $M_{app} = (evenlist, evenlist, \top)$, then we would infer the set of all lists as the input type for *x*, i.e., $I_{app} = (list, evenlist, \top)$ (note that there are recursive calls to *append* with lists of odd length).

We obtain (1) and (2) as the special case of (3) where the lower bounds for all input types but *main* are given as the empty set. Hence, it is sufficient to formulate the type inference only for the case (3).

Definition 10 (Inference of the set-based directional type $\mathcal{T}_{sb}(\mathcal{P}, (M_p)_{p \in \text{Pred}})$). *Given a program \mathcal{P} and a family of lower bounds M_p for the input types of the predicates p of the program, we infer the set-based directional type*

$$\mathcal{T}_{sb}(\mathcal{P}, (M_p)_{p \in \text{Pred}}) = (\text{Sat}(I_p) \rightarrow \text{Sat}(O_p))_{p \in \text{Pred}}$$

where I_p and O_p are the denotations of the predicates p^{In} and p^{Out} in the least model of the program

$$\mathcal{P}_{InOut}^\# \cup \{p^{In}(x) \leftarrow M_p(x) \mid p \in \text{Pred}\}. \quad (2)$$

The definition of type inference above leaves open in which formalism the lower bounds are specified and how the inferred directional types are presented to the user. There is a wide variety of formalisms that coincide in the expressive power of regular sets of trees and that can be effectively translated one into another and, hence, for which our type inference yields an effective procedure.

More concretely, we propose to represent the lower bounds M_p through logic programs in restricted syntax (see Sect. 2) that corresponds directly to alternating tree automata (and also to the positive set expressions considered in 11; an even more restricted syntax corresponds to non-deterministic tree automata and to positive set expressions without intersection). We attach these logic programs to the program in (2) as definitions of the sets M_p . Then, we can apply one of the known algorithms (see, e.g., [19, 18, 11, 10]) in order to compute, in single-exponential time (in the size of the program \mathcal{P} and the programs for the sets M_p), a logic program that corresponds to a non-deterministic tree automaton and that is equivalent to the program in (2) wrt. the least model and, thus, represents $\mathcal{T}_{sb}(\mathcal{P}, (M_p)_{p \in \text{Pred}})$. The representation of sets by a non-deterministic tree automaton is a good representation in the sense that, for example, the test of emptiness can be done in linear time.

If, in Definition 10, we replace $\mathcal{P}_{InOut}^\#$ with \mathcal{P}_{InOut} , then we obtain the uniformly most precise directional type $\mathcal{T}_{min}(\mathcal{P})$ wrt. given lower bounds for input types. In general, we cannot effectively compute $\mathcal{T}_{min}(\mathcal{P})$ (e.g., test emptiness of its input and output types).

We repeat that the following comparison of the set-based directional type of \mathcal{P} with discriminative directional types of \mathcal{P} is interesting because these form the subclass of directional types for which the type check in 11 is sound and complete.

Theorem 3 (Soundness and Quality of Type Inference). *The program \mathcal{P} is well-typed wrt. the set-based directional type $\mathcal{T}_{sb}(\mathcal{P}, (M_p)_{p \in \text{Pred}})$. Moreover, this type is uniformly more precise than every discriminative directional type of \mathcal{P} whose family of input types contains $(M_p)_{p \in \text{Pred}}$.*

Proof. The two statements are direct consequences of Propositions 11 and 14 respectively, together with Theorem 11. \square

5 Complexity of Directional Type Checking

Theorem 4. *The complexity of directional type checking of logic programs for discriminative types is DEXPTIME-complete.*

Proof. The DEXPTIME-hardness follows from refining the argument in the proof of Theorem 18 in [1] with the two facts that (1) discriminative types are denoted by path closed sets, and (2) testing the non-emptiness of a sequence of n tree automata is DEXPTIME-hard even if the automata are restricted to deterministic ones (which recognize exactly path closed sets) [29].

We obtain a type checking algorithm in single-exponential time as follows. The input is the program \mathcal{P} and the discriminative directional type

$$\mathcal{T} = (\text{Sat}(I_p) \rightarrow \text{Sat}(O_p))$$

where all types I_p and O_p are given by ground set expressions (a special case of which are regular tree expressions or non-deterministic tree automata). Ground set expressions correspond to alternating tree automata which are self-dual; i.e., we can obtain ground set expressions \tilde{I}_p and \tilde{O}_p representing the complement by a syntactic transformation in linear time. We can translate ground expressions into logic programs defining predicates p_{I_p} and p_{O_p} such that they denote I_p and O_p wrt. the least-model semantics (and similarly predicates $p_{\tilde{I}_p}$ and $p_{\tilde{O}_p}$ for \tilde{I}_p and \tilde{O}_p).

We now use one of the well-known single-exponential time algorithms [19,18,11,10] to compute (the non-deterministic tree automaton representing) the least model of the set-based abstraction $\mathcal{P}^\#$ of a logic program \mathcal{P} . We apply such an algorithm to the program $\mathcal{P}_{InOut}^\#(\mathcal{T})$ that we obtain from $\mathcal{P}_{InOut}^\#$ by adding the clauses $p^{In}(x) \leftarrow p_{I_p}(x)$ and $p^{Out}(x) \leftarrow p_{O_p}(x)$ and the logic programs defining p_{I_p} and p_{O_p} .

We use the result in order to test whether the denotations of p^{In} and p^{Out} under the least model of the program $\mathcal{P}_{InOut}(\mathcal{T})$ are exactly I_p and O_p . This holds if and only if \mathcal{T} is a directional type of \mathcal{P} (otherwise, we have a proper inclusion for at least one p ; see the correctness proof below). The test works by testing whether the intersection of (the non-deterministic tree automaton representing) the denotations of p^{In} and p^{Out} under the least model of the program $\mathcal{P}_{InOut}(\mathcal{T})$ with the complements $p_{\tilde{I}_p}$ and $p_{\tilde{O}_p}$ is empty. This can be done in one pass by taking the conjunction of $\mathcal{P}_{InOut}(\mathcal{T})$ with the logic programs defining $p_{\tilde{I}_p}$ and $p_{\tilde{O}_p}$ and testing the emptiness of new predicates defined as the intersection of p_{I_p} and $p_{\tilde{I}_p}$ (and p_{O_p} and $p_{\tilde{O}_p}$).

The correctness of the algorithm follows with Theorem 1 and Proposition 1 and 4. In detail: Given a discriminative directional type \mathcal{T} and a program \mathcal{P} , we have that \mathcal{P} is well-typed wrt. \mathcal{T} iff the corresponding subset $\mathcal{M}_{\mathcal{T}}$ of the Herbrand base is a (path closed) model of \mathcal{P}_{InOut} by Theorem 1. If this is the case then $\mathcal{M}_{\mathcal{T}}$ is equal to the least model \mathcal{M}_0 of $\mathcal{P}_{InOut}(\mathcal{T})$, since $\mathcal{M}_{\mathcal{T}} \subseteq \mathcal{M}_0$ by definition of $\mathcal{P}_{InOut}(\mathcal{T})$, and $\mathcal{M}_{\mathcal{T}} \supseteq \mathcal{M}_0$ holds by Proposition 4 (note that $\mathcal{M}_{\mathcal{T}}$

is a path closed model of \mathcal{P}_{InOut} and of the additional clauses translating \mathcal{T} and, thus, contains the least one). On the other hand, if $\mathcal{M}_{\mathcal{T}}$ is equal to the least model of $\mathcal{P}_{InOut}^{\#}(\mathcal{T})$, then \mathcal{T} is a directional type by Proposition 11 and Theorem 12. \square

We note that the procedure above yields a semi-test (in the same sense as the one in 11) for well-typedness wrt. the class of *general* directional types, since the equivalence between the inferred type and the given one is a sufficient (but generally not necessary) condition for well-typedness wrt. the given type. We repeat that it implements a full test wrt. discriminative directional types.

6 Future Work

One of the obvious directions for future work is an implementation. We already have a working prototype implementation on top of the saturation-based theorem prover SPASS [30]. The first results are very promising; due to specific theorem-proving techniques like powerful redundancy criteria, one obtains a decision procedure for the emptiness test that is quite efficient on our examples. We ran our prototype on the benchmark programs from [9]. We were able to detect emptiness of the main predicate for six of the twelve programs, while a similar method of Gallagher succeeded in three cases. We believe that by using the tree-automata techniques suggested in [18] together with automata minimization (and conversion to the syntax of ground set expressions), we can further improve the efficiency of our implementation and the readability of the output.

Acknowledgments. We thank David McAllester for turning us on to magic sets and thereby to directional types. We thank Harald Ganzinger for useful comments.

References

1. A. Aiken and T. K. Lakshman. Directional type checking of logic programs. In B. L. Charlier, editor, *1st International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 43–60, Namur, Belgium, Sept. 1994. Springer Verlag.
2. K. R. Apt. Declarative programming in Prolog. In D. Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 12–35, Vancouver, Canada, 1993. The MIT Press.
3. K. R. Apt. Program verification and Prolog. In E. Börger, editor, *Specification and Validation methods for Programming languages and systems*, pages 55–95. Oxford University Press, 1995.
4. K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. M. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium*, volume 711 of *lncs*, pages 1–19, Gdansk, Poland, 30 Aug.– 3 Sept. 1993. Springer.

5. J. Boye. *Directional Types in Logic Programming*. PhD thesis, Department of Computer and Information Science, Linköping University, 1996.
6. J. Boye and J. Maluszynski. Two aspects of directional types. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 747–764, Cambridge, June 13–18 1995. MIT Press.
7. J. Boye and J. Maluszynski. Directional types and the annotation method. *Journal of Logic Programming*, 33(3):179–220, Dec. 1997.
8. F. Bronsard, T. K. Lakshman, and U. S. Reddy. A framework of directionality for proving termination of logic programs. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 321–335, Washington, USA, 1992. The MIT Press.
9. M. Bruynooghe, H. Vandecasteele, A. de Waal, and M. Denecker. Detecting unsolvable queries for definite logic programs. In *Proceedings of the Symposium on Programming Language Implementation and Logic Programming (PLILP'98)*, LNCS. Springer-Verlag, 1998. to appear.
10. W. Charatonik, D. McAllester, D. Niwiński, A. Podelski, and I. Walukiewicz. The Horn mu-calculus. To appear in Vaughan Pratt, editor, *Proceedings of the 13th IEEE Annual Symposium on Logic in Computer Science*.
11. W. Charatonik, D. McAllester, and A. Podelski. Computing the least and the greatest model of the set-based abstraction of logic programs. Presented at the Dagstuhl Workshop on Tree Automata, October 1997.
12. W. Charatonik and L. Pacholski. Negative set constraints with equality. In *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 128–136, 1994.
13. W. Charatonik and A. Podelski. Set constraints for greatest models. Technical Report MPI-I-97-2-004, Max-Planck-Institut für Informatik, April 1997. www.mpi-sb.mpg.de/~podelski/papers/greatest.html.
14. M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In B. L. Charlier, editor, *Proceedings of the First International Static Analysis Symposium*, Lecture Notes in Computer Science 864, pages 281–296. Springer Verlag, 1994.
15. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dam, editor, *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of LNCS, pages 22–50. Springer-Verlag, June 1996.
16. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proc. POPL '92*, pages 83–94. ACM Press, 1992.
17. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Record of FPCA '95 - Conference on Functional Programming and Computer Architecture*, pages 170–181, La Jolla, California, USA, 25–28 June 1995. SIGPLAN/SIGARCH/WG2.8, ACM Press, New York, USA.
18. P. Devienne, J.-M. Talbot, and S. Tison. Set-based analysis for logic programming and tree automata. In *Proceedings of the Static Analysis Symposium, SAS'97*, volume 1302 of LNCS, pages 127–140. Springer-Verlag, 1997.
19. T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, July 1991.
20. J. Gallagher and D. A. de Waal. Regular approximations of logic programs and their uses. Technical Report CSTR-92-06, Department of Computer Science, University of Bristol, 1992.

21. J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. V. Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, Santa Margherita Ligure, Italy, 1994. The MIT Press.
22. F. Gécseg and M. Steinby. *Tree Automata*. Akademiai Kiado, 1984.
23. N. Heintze. Practical aspects of set based analysis. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 765–779, Washington, USA, 1992. The MIT Press.
24. N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 197–209, January 1990.
25. N. Heintze and J. Jaffar. Set constraints and set-based analysis. In *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, LNCS 874, pages 281–298. Springer-Verlag, 1994.
26. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables. *Journal of Logic Programming*, 13(2-3):205–258, 1992.
27. P. Mishra. Towards a theory of types in Prolog. In *IEEE International Symposium on Logic Programming*, pages 289–298, 1984.
28. Y. Rouzard and L. Nguyen-Phuong. Integrating modes and subtypes into a Prolog type-checker. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 85–97, Washington, USA, 1992. The MIT Press.
29. H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52:57–60, 1994.
30. C. Weidenbach. Spass version 0.49. *Journal of Automated Reasoning*, 18(2):247–252, 1997.
31. E. Yardeni and E. Shapiro. A type system for logic programs. In E. Shapiro, editor, *Concurrent Prolog*, volume 2, chapter 28, pages 211–244. The MIT Press, 1987.

Finite Subtype Inference with Explicit Polymorphism

Dominic Duggan

Department of Computer Science,
Stevens Institute of Technology,
Castle Point on the Hudson,
Hoboken, New Jersey 07030.
dduggan@cs.stevens-tech.edu

Abstract. *Finite subtype inference* occupies a middle ground between Hindley-Milner type inference (as in ML) and subtype inference with recursively constrained types. It refers to subtype inference where only finite types are allowed as solutions. This approach avoids some open problems with general subtype inference, and has practical motivation where recursively constrained types are not appropriate. This paper presents algorithms for finite subtype inference, including checking for entailment of inferred types against explicitly declared polymorphic types. This resolves for finite types a problem that is still open for recursively constrained types. Some motivation for this work, particularly for finite types and explicit polymorphism, is in providing subtype inference for first-class container objects with polymorphic methods.

1 Introduction

Type inference is the process of statically type-checking a program where some or all of the type information has been omitted from the program text. ML and Haskell are examples of programming languages where type inference has been a spectacular success. The particular flavor of type inference used by ML and Haskell is *Hindley-Milner type inference* [14]. The type-checker accumulates equality constraints via a tree walk of the abstract syntax tree, and then uses a unification algorithm to compute a (most general) unifying substitution for these constraints.

More recently attention has been focused on *subtype inference* [23,177,1922]. With this work, the type-checker accumulates subtype constraints while traversing the abstract syntax tree, and then applies a constraint solver to check these constraints for consistency. Pottier [19] and Smith and Trifonov [22] have considered the problem of entailment in these type systems, which is important for example in interface matching. Subtype inference continues to be an important avenue of research, particularly in simplifying inferred types to make them practically useful.

Hindley-Milner type inference and subtype inference represent two extremes in the type inference continuum:

Hindley-Milner	Finite Subtype	Subtype
Equality	Subtyping	Subtyping
Finite types	Finite types	Infinite types
Inferred monotypes	Inferred monotypes	Inferred monotypes
Inferred polytypes	$\left\{ \begin{array}{l} \text{Inferred polytypes} \\ \text{Specified polytypes}^* \end{array} \right\}$	Inferred polytypes

Between these two extremes, there is an intermediate point: *finite subtype inference*. While this alternative allows subtyping and type subsumption, it does not assume that types are potentially infinite trees (as with the most recent work on subtype inference).

Why should we consider subtype inference with finite types? It is worth recalling why ML for example does not allow circular types (types as potentially infinite trees). The problem was pointed out by Solomon [21]: the problem of deciding the equality of parameterized recursive types is equivalent to the problem of deciding the equality of deterministic context-free languages (DCFLs), which is still after several decades an open problem. This problem is avoided in ML type inference by making the folding and unfolding of recursive types explicit (using data constructors and pattern-matching, respectively), so that circular types are not needed.

A motivation for infinite types in subtype inference is to support objects with recursive interfaces. However the problem discovered by Solomon also holds for recursive interfaces for container objects. Consider for example a `set` object with interface:

$$\begin{aligned} \text{set}(\alpha) = \{ & \text{map} : \forall \beta. (\alpha \rightarrow \beta) \rightarrow \text{set}(\beta), \\ & \text{product} : \forall \beta. \text{set}(\beta) \rightarrow \text{set}(\alpha * \beta), \\ & \text{power} : \forall \beta. \text{unit} \rightarrow \text{set}(\text{set}(\alpha)) \end{aligned}$$

All of the methods in this interface are examples of non-regular recursion in the object interface. In a companion paper [5], we have developed an object design for ML-like languages that avoids this problem. The approach there is to again make the folding and unfolding of recursive object interfaces explicit, in object creation and method invocation, respectively. This work introduces the possibility that circular types for recursive object interfaces, while useful for simple objects, may not be so useful for container objects.

Early work on subtype inference considered atomic subtyping with finite types [15, 12, 9]. However there are non-trivial differences between finite subtype inference with atomic subtyping and with record containment. For example even checking for finiteness is markedly different, as elaborated upon in Sect. 5.

Another design point is whether polymorphic types should be inferred or specified. All of the work so far assumes that polymorphic types are inferred. The disadvantage of these approaches is that the inferred types are large and complex, diminishing their practical usefulness, despite recent work on simplifying inferred types [7, 19, 22]. One way to avoid this problem is to require that the programmer provide explicit interfaces for polymorphic functions. This approach introduces fresh technical complications of its own. In Hindley-Milner type inference, *mixed-prefix unification* has been used to control scoping of type variables with explicit polymorphic type declarations (an idea originally used by Leroy and Mauny [11], and subsequently rediscovered by Odersky and Läufer

[16]). In this paper we extend subtype inference with *constraint-solving under a mixed prefix*, in order to support subtype inference with *explicit polymorphism*.

Explicit polymorphism also derives motivation from our work on container objects with recursive interfaces [5]. We avoid the problems with first-class polymorphism in Hindley-Milner type inference, by requiring explicit type specifications on polymorphic methods. This is similar to the use of universal types to incorporate impredicativity into Hindley-Milner type inference [20, 16, 10], but tied to the object system instead of to datatypes (again because we are concerned with container objects with polymorphic methods).

Even if we are not concerned with polymorphic methods, explicit polymorphism is required if we wish to provide a type-checking algorithm for the Abadi and Cardelli object type system, for example [1]. In that type system, the type rule for method update is structural [1, Sect. 16.2], meaning that the new method definition must be parametric in the type of self (the type of self is a type parameter constrained by the object interface).

Explicit polymorphism requires that it be possible to check for entailment of inferred types from declared types. For infinite types this is problematic. Although incomplete algorithms have been published [19, 22], the decidability of entailment remains open [22]. In this paper we demonstrate that entailment is decidable for finite subtyping, giving further motivation for our approach.

Sect. 2 introduces our type system. We do not overburden the paper with any details of the object system mentioned earlier [5], but present a familiar ML-like language with record-based subtyping and explicit polymorphism. Sect. 3 provides the type inference algorithm. Sect. 4 provides algorithms for checking consistency and entailment; these algorithms must be defined mutually recursively. Sect. 5 considers the check for finite solutions; perhaps surprisingly, this check must be done after consistency and entailment checking. Sect. 6 considers the use of mixed prefix constraint-solving to check that there are well-scoped solutions. Finally Sect. 8 considers further related work and provides conclusions.

2 Type System

The mini-language we consider is a language with functions, pairs and records. Subtyping is based on containment between record types. This is extended contravariantly to function types and covariantly to product types. Polymorphic types allow quantified type variables to be constrained by upper bounds. We use $\forall \alpha_n <: \tau_n. \tau$ generically for a sequence of quantifiers where all, none or some of the variables may have upper bounds.

$$\begin{aligned}
 \tau &::= \alpha \mid \mathbf{t} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * \tau_2 \mid \{l_1 : \tau_1, \dots, l_n : \tau_n\} \\
 \sigma &::= \forall \alpha. \sigma \mid \forall \alpha <: \tau. \sigma \mid \tau \\
 e &::= x \mid \lambda x. e \mid (e_1 \ e_2) \mid (e_1, e_2) \mid \mathbf{fst} \ e \mid \mathbf{snd} \ e \mid \\
 &\quad \mathbf{let} \ x : \sigma = e_1 \ \mathbf{in} \ e_2 \mid \{l_1 = e_1, \dots, l_n = e_n\} \mid e.l
 \end{aligned}$$

The type rules are specified in Fig. 1 using judgements of the form $\Gamma; A \vdash e : \tau$. A is a sequence of program variable bindings ($x : \sigma$), while Γ is a sequence of type

variable bindings ($(\alpha <: \tau)$, or just α where there is no upper bound specified in a type annotation).

The main construct of interest is the `let` construct. This allows generalization of types to polymorphic types, but requires an explicit type annotation. This is demonstrated by the `LET` type rule in Fig. 1. It is also possible to define a monomorphic version of the `let`, that does not require a type annotation:

$$\text{letmono } x = e_1 \text{ in } e_2 \equiv (\lambda x. e_2) e_1$$

$\frac{A(x) = \forall \overline{\alpha_n} <: \overline{\tau_n}. \tau \quad \Gamma \vdash \tau'_i <: \{\overline{\tau'_{i-1}} / \overline{\alpha_{i-1}}\} \tau_i \text{ for } i = 1, \dots, n}{\Gamma; A \vdash x : \{\overline{\tau'_n} / \overline{\alpha_n}\} \tau}$	(VAR)
$\frac{\Gamma; A \vdash e_1 : (\tau_2 \rightarrow \tau_1) \quad \Gamma; A \vdash e_2 : \tau_2}{\Gamma; A \vdash (e_1 e_2) : \tau_1}$	(APP)
$\frac{\Gamma; A, x : \tau_1 \vdash e : \tau_2}{\Gamma; A \vdash (\lambda x. e) : (\tau_1 \rightarrow \tau_2)}$	(ABS)
$\frac{\Gamma; A \vdash e_1 : \tau_1 \quad \Gamma; A \vdash e_2 : \tau_2}{\Gamma; A \vdash (e_1, e_2) : (\tau_1 * \tau_2)}$	(PAIR)
$\frac{\Gamma; A \vdash e : (\tau_1 * \tau_2)}{\Gamma; A \vdash \mathbf{fst} e : \tau_1}$	(FST)
$\frac{\Gamma; A \vdash e : (\tau_1 * \tau_2)}{\Gamma; A \vdash \mathbf{snd} e : \tau_2}$	(SND)
$\frac{\Gamma; A \vdash e_1 : \tau_1 \quad \dots \quad \Gamma; A \vdash e_n : \tau_n}{\Gamma; A \vdash \{l_1 = e_1, \dots, l_n = e_n\} : \{l_1 : \tau_1, \dots, l_n : \tau_n\}}$	(RECORD)
$\frac{\Gamma; A \vdash e : \{l : \tau\}}{\Gamma; A \vdash e.l : \tau}$	(SELECT)
$\frac{\Gamma, \overline{\alpha_n} <: \overline{\tau_n}; A \vdash e_1 : \tau_1 \quad \Gamma; A, x : (\forall \overline{\alpha_n} <: \overline{\tau_n}. \tau_1) \vdash e_2 : \tau_2}{\Gamma; A \vdash (\mathbf{let} \ x : (\forall \overline{\alpha_n} <: \overline{\tau_n}. \tau_1) = e_1 \text{ in } e_2) : \tau_2}$	(LET)
$\frac{\Gamma; A \vdash e : \tau \quad \Gamma \vdash \tau <: \tau'}{\Gamma; A \vdash e : \tau'}$	(SUB)

Fig. 1. Type Rules

Although we are primarily concerned with finite types and finite solutions to constraint sets, we will need to be able to reason about infinite solutions to constraint sets. For reasons of space, we omit the details and refer to Pottier [19]. We only summarize that a type (tree) can be considered as a mapping from *paths* π to types (subterms). Paths are sequences from the alphabet $\{d, r, f, s\} \cup \text{FieldName}$. d and r denote the domain and range, respectively, of a function type, while f and s denote the first and second

$\frac{\Gamma = \Gamma_1, \alpha <: \tau, \Gamma_2}{\Gamma \vdash \alpha <: \tau}$	(SUBAX)
$\Gamma \vdash \tau <: \tau$	(SUBREΦ)
$\frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3}$	(SUBTRANS)
$\frac{\Gamma \vdash \tau'_1 <: \tau_1 \quad \Gamma \vdash \tau_2 <: \tau'_2}{\Gamma \vdash (\tau_1 \rightarrow \tau_2) <: (\tau'_1 \rightarrow \tau'_2)}$	(SUBFUN)
$\frac{\Gamma \vdash \tau_1 <: \tau'_1 \quad \Gamma \vdash \tau_2 <: \tau'_2}{\Gamma \vdash (\tau_1 * \tau_2) <: (\tau'_1 * \tau'_2)}$	(SUBPAIR)
$\frac{\Gamma \vdash \tau_i <: \tau'_i \text{ for } i = 1, \dots, n \text{ where } m \geq n}{\Gamma \vdash \{\overline{l_m} : \overline{\tau_m}\} <: \{\overline{l_n} : \overline{\tau_n}\}}$	(SUBREC)

Fig. 2. Subtype Rules

component types, respectively, of a product type. $\text{dom}(\tau) = \{\pi \mid \tau(\pi) \text{ is defined}\}$. A type τ is finite if $\text{dom}(\tau)$ is finite, and infinite otherwise.

3 Type Inference

The type inference algorithm is provided in Fig. 3. It uses judgements of the form $\mathcal{Q}; A \vdash e : \tau \mid \mathcal{Q}', C$. \mathcal{Q} and \mathcal{Q}' are *quantifier prefixes*, while C is a constraint set, described by:

$$\begin{aligned} \mathcal{Q} &::= \exists \alpha \mid \forall \alpha \mid \forall \alpha <: \tau \mid \mathcal{Q}_1. \mathcal{Q}_2 \\ C &::= \{ \} \mid \{ \tau_1 <: \tau_2 \} \mid C_1 \cup C_2 \end{aligned}$$

The quantifier prefix records “flexible” (existential) and “rigid” (universal) variables introduced during type inference, while the relative ordering records the scope of the type variables (existential α can be instantiated to a type containing universal β only if β is quantified to the left of α in the quantifier prefix).

The inputs to the algorithm are \mathcal{Q} , A and e . The outputs are the inferred type τ , a set of constraints C constraining instantiations of A and τ , and an extension \mathcal{Q}' of the quantifier prefix \mathcal{Q} .

We will need to reason about satisfying substitutions for constraints under a mixed prefix. We use the following technical device (introduced by Duggan [6], and extending the unification logic originally introduced by Miller [13]):

Definition 1 (Constraint Logic With Substitutions). A term \mathcal{F} of the constraint logic is a pair $\mathcal{Q}C$ where \mathcal{Q} is a quantifier prefix, and where C is a set of constraints. Derivability for the judgement form $\theta \models \Gamma \vdash \mathcal{F}$ is defined by the following rules:

$$\frac{\Gamma \vdash C}{\theta \models \Gamma \vdash C}$$

$\frac{A(x) = \forall \overline{\alpha_m} <: \overline{\tau_m}. \tau \quad \mathcal{Q}' = \mathcal{Q}, \exists \overline{\alpha_m}}{\mathcal{Q}; A \vdash x : \tau \mid \mathcal{Q}', \{\alpha_i <: \tau_i \mid i = 1, \dots, m\}}$	(VAR)
$\frac{\mathcal{Q}, \exists \alpha; A, x : \alpha \vdash e : \tau \mid \mathcal{Q}', C}{\mathcal{Q}; A \vdash (\lambda x. e) : (\alpha \rightarrow \tau) \mid \mathcal{Q}', C}$	(ABS)
$\frac{\mathcal{Q}; A \vdash e_1 : \tau_1 \mid \mathcal{Q}_1, C_1 \quad \mathcal{Q}_1; A \vdash e_2 : \tau_2 \mid \mathcal{Q}_2, C_2}{\mathcal{Q}; A \vdash (e_1 e_2) : \alpha \mid (\mathcal{Q}_2, \exists \alpha), C_1 \cup C_2 \cup \{\tau_1 <: (\tau_2 \rightarrow \alpha)\}}$	(APP)
$\frac{\mathcal{Q}; A \vdash e_1 : \tau_1 \mid \mathcal{Q}_1, C_1 \quad \mathcal{Q}_1; A \vdash e_2 : \tau_2 \mid \mathcal{Q}_2, C_2}{\mathcal{Q}; A \vdash (e_1, e_2) : (\tau_1 * \tau_2) \mid \mathcal{Q}_2, C_1 \cup C_2}$	(PAIR)
$\frac{\mathcal{Q}; A \vdash e : \tau \mid \mathcal{Q}', C}{\mathcal{Q}; A \vdash (\mathbf{fst} e) : \alpha \mid (\mathcal{Q}', \exists \alpha, \exists \beta), C \cup \{\tau <: (\alpha * \beta)\}}$	(FST)
$\frac{\mathcal{Q}; A \vdash e : \tau \mid \mathcal{Q}', C}{\mathcal{Q}; A \vdash (\mathbf{snd} e) : \beta \mid (\mathcal{Q}', \exists \alpha, \exists \beta), C \cup \{\tau <: (\alpha * \beta)\}}$	(SND)
$\frac{\mathcal{Q}_0 = \mathcal{Q} \quad \mathcal{Q}_{i-1}; A \vdash e_i : \tau_i \mid \mathcal{Q}_i, C_i \text{ for } i = 1, \dots, n}{\mathcal{Q}; A \vdash \{l_1 = e_1, \dots, l_n = e_n\} : \{l_1 : \tau_1, \dots, l_n : \tau_n\} \mid \mathcal{Q}_n, C_1 \cup \dots \cup C_n}$	(RECORD)
$\frac{\mathcal{Q}; A \vdash e : \tau \mid \mathcal{Q}', C}{\mathcal{Q}; A \vdash e.l : \alpha \mid (\mathcal{Q}', \exists \alpha), C \cup \{\tau <: \{l : \alpha\}\}}$	(SELECT)
$\frac{(\mathcal{Q}, \forall \overline{\alpha_n} <: \overline{\tau_n}); A \vdash e_1 : \tau_1 \mid \mathcal{Q}_1, C_1 \quad \mathcal{Q}_1; A, x : (\forall \overline{\alpha_n} <: \overline{\tau_n}. \tau) \vdash e_2 : \tau_2 \mid \mathcal{Q}_2, C_2}{\mathcal{Q}; A \vdash (\mathbf{let} x : (\forall \overline{\alpha_n} <: \overline{\tau_n}. \tau) = e_1 \text{ in } e_2) : \tau_2 \mid \mathcal{Q}_2, C_1 \cup C_2 \cup \{\tau_1 <: \tau\}}$	(LET)

Fig. 3. Type Inference

$$\frac{\alpha \notin \text{dom}(\theta) \cup \text{dom}(\Gamma) \quad \theta \models \Gamma, \alpha <: \tau \vdash \mathcal{F}}{\theta \models \Gamma \vdash (\forall \alpha <: \tau. \mathcal{F})}$$

$$\frac{\alpha \in \text{dom}(\theta) \quad FV(\theta(\alpha)) \subseteq \text{dom}(\Gamma) \quad \theta \models \Gamma \vdash \{\theta(\alpha)/\alpha\} \mathcal{F}}{\theta \models \Gamma \vdash (\exists \alpha : \chi. \mathcal{F})}$$

Unification logic was originally introduced by Miller [13] in order to reason about the correctness of unification under a mixed prefix. Miller used a complicated mechanism based on decomposing a substitution into the composition of a sequence of single-variable substitutions. Our construction takes a simpler approach, essentially applying the ideas of the Curry-Howard isomorphism to extend the unification logic to a type system for reasoning about the correctness of satisfying substitutions. In particular the third rule requires that, in the substitution for an existential variable α , this substitution only contains as free variables those variables that are universally quantified to the left of α .

Definition 2 (Satisfiability). *Given a quantifier prefix \mathcal{Q} and constraint formula C . θ satisfies \mathcal{Q} and C if $\theta \models \{\} \vdash \mathcal{Q}C$. Denote this by $\theta \models \mathcal{Q}C$. \mathcal{Q} and C are satisfiable if*

there is some substitution θ such that $\theta \models QC$. Denote that Q and C are satisfiable by $\models QC$.

4 Containment and Entailment

The first step in constraint-checking is to verify that there is a satisfying substitution for the accumulated constraints, ignoring finiteness and well-scoping in the satisfying substitution. We need to check that the constraints are consistent, and are entailed by the hypothetical bounds on the universally quantified variables.

Definition 3 (Entailment Algorithm). *The containment and entailment algorithm is provided in Fig. 4. Define $QC \xRightarrow{*} Q'C'$ to denote the repeated application of the algorithm to QC until it reaches a fixed point $Q'C'$. Let $QC \downarrow_{<}$ denote $Q'C'$, the result of applying the algorithm to QC .*

Rules (1)-(4) are the usual subtype closure, combining transitive closure with downward closure conditions, and check for containment in the original constraint set. The latter conditions must be satisfied for the original subtype constraints to be satisfied.

The remaining transitions check for entailment of the accumulated constraints from the hypotheses. Rule (5) checks that an upper bound on a universal variable can be satisfied. Rules (6) and (7) check for compatibility of upper and lower bounds on existential variables. “Compatibility” means that we do not have incompatible non-variable bounds on variables, such as $((\tau_1 * \tau_2) <: \beta), ((\tau'_1 \rightarrow \tau'_2) <: \beta) \in C$. We also need to check for compatibility of universal variables with other types in lower and upper bounds, for example $((\alpha * \tau_1) <: \beta), ((\tau' * \tau_2) <: \beta) \in C$ where α is universal (with upper bound τ) and β existential.

The predicates $\tau \sqcap \tau'$ and $\tau \sqcup \tau'$ are used to check for compatibility. They are generated by Rules (6) and (7), and analyzed by Rules (8)-(17). Rules (15) and (16) are used to check compatibility of universal variables with non-variable types. For example, given:

$$Q = Q_1. \forall \alpha <: \tau. Q_2 \text{ and } C \supseteq \{\alpha <: \beta, \tau' <: \beta\}$$

where β is existential and τ' non-variable, compability of the latter two constraints requires that $(\tau <: \beta)$ and $(\tau' <: \beta)$ be compatible, as checked by Rule (15). For Rule (16), compatibility of $(\beta <: \alpha)$ and $(\beta <: \tau')$ requires that $\tau <: \tau'$. Rule (17) uses the definition:

Definition 4. *Define $Q \vdash \alpha <: \beta$ if one of the following holds:*

1. $\alpha = \beta$; or
2. $Q = Q_1. \forall \alpha <: \beta. Q_2$; or
3. $Q \vdash \alpha <: \gamma$ and $Q \vdash \gamma <: \beta$ for some γ .

Define $Q \vdash \alpha \sqcap \beta$ to be: $Q \vdash \alpha <: \beta$ or $Q \vdash \beta <: \alpha$.

Rule (5) and Rule (16) are noteworthy for the fact that they generate new subtype constraints, and this is the reason that the containment and entailment algorithms must be defined mutually recursively. Consider for example:

$$\forall \alpha <: (\{x : \text{int}\} * \{x : \text{int}\}). \exists \beta. \exists \gamma. \{\alpha <: (\beta * \gamma), \beta <: \{x : \text{int}\}, \gamma <: \{y : \text{int}\}\}$$

Entailment generates the constraint:

$$(\{x : \text{int}\} * \{x : \text{int}\}) <: (\beta * \gamma)$$

and containment generates the constraints:

$$\{x : \text{int}\} <: \beta, \{x : \text{int}\} <: \gamma, \{x : \text{int}\} <: \{x : \text{int}\}, \{x : \text{int}\} <: \{y : \text{int}\}$$

and the last of these constraints violates containment.

$$QC \implies Q(C \cup \{(\tau_1 <: \tau_2)\}) \text{ if } (\tau_1 <: \alpha), (\alpha <: \tau_2) \in C \quad (1)$$

$$QC \implies Q(C \cup \{(\tau'_1 <: \tau_1), (\tau_2 <: \tau'_2)\}) \text{ if } ((\tau_1 \rightarrow \tau_2) <: (\tau'_1 \rightarrow \tau'_2)) \in C \quad (2)$$

$$QC \implies Q(C \cup \{(\tau_1 <: \tau'_1), (\tau_2 <: \tau'_2)\}) \text{ if } ((\tau_1 * \tau_2) <: (\tau'_1 * \tau'_2)) \in C \quad (3)$$

$$QC \implies Q(C \cup \{(\tau_i <: \tau'_i) \mid i = 1, \dots, n\}) \text{ if } (\{\overline{l}_m : \overline{\tau}_m\} <: \{\overline{l}_n : \overline{\tau}'_n\}) \in C \text{ and } m \geq n \quad (4)$$

$$QC \implies Q(C \cup \{\tau <: \tau'\}) \text{ if } Q = Q_1 \forall \alpha <: \tau Q_2 \text{ and } (\alpha <: \tau') \in C, \tau' \notin EV(Q) \quad (5)$$

$$QC \implies Q(C \cup \{\tau \sqcap \tau'\}) \text{ if } (\tau <: \beta), (\tau' <: \beta) \in C, \beta \in EV(Q), \tau, \tau' \notin EV(Q) \quad (6)$$

$$QC \implies Q(C \cup \{\tau \sqcup \tau'\}) \text{ if } (\beta <: \tau), (\beta <: \tau') \in C, \beta \in EV(Q), \tau, \tau' \notin EV(Q) \quad (7)$$

$$QC \implies Q(C \cup \{\tau'_1 \sqcup \tau_1, \tau_2 \sqcap \tau'_2\}) \text{ if } ((\tau_1 \rightarrow \tau_2) \sqcap (\tau'_1 \rightarrow \tau'_2)) \in C \quad (8)$$

$$QC \implies Q(C \cup \{\tau_1 \sqcap \tau'_1, \tau_2 \sqcap \tau'_2\}) \text{ if } ((\tau_1 * \tau_2) \sqcap (\tau'_1 * \tau'_2)) \in C \quad (9)$$

$$QC \implies Q(C \cup \{\tau'_1 \sqcap \tau_1, \tau_2 \sqcup \tau'_2\}) \text{ if } ((\tau_1 \rightarrow \tau_2) \sqcup (\tau'_1 \rightarrow \tau'_2)) \in C \quad (10)$$

$$QC \implies Q(C \cup \{\tau_1 \sqcup \tau'_1, \tau_2 \sqcup \tau'_2\}) \text{ if } ((\tau_1 * \tau_2) \sqcup (\tau'_1 * \tau'_2)) \in C \quad (11)$$

$$QC \implies Q(C \cup \{(\tau_i \sqcup \tau'_j) \mid l_i = l'_j\}) \text{ if } (\{\overline{l}_m : \overline{\tau}_m\} \sqcup \{\overline{l}_n : \overline{\tau}'_n\}) \in C \quad (12)$$

$$QC \implies Q(C \cup \{\tau \sqcap \tau'\}) \text{ if } \alpha \in EV(Q) \text{ and } (\alpha \sqcap \tau'), (\tau <: \alpha) \in C \quad (13)$$

$$QC \implies Q(C \cup \{\tau \sqcup \tau'\}) \text{ if } \alpha \in EV(Q) \text{ and } (\alpha \sqcup \tau'), (\alpha <: \tau) \in C \quad (14)$$

$$QC \implies Q(C \cup \{\tau \sqcap \tau'\}) \text{ if } Q = Q_1 \forall \alpha <: \tau Q_2 \text{ and } (\alpha \sqcap \tau') \in C, \tau' \notin TyVar \quad (15)$$

$$QC \implies Q(C \cup \{\tau <: \tau'\}) \text{ if } Q = Q_1 \forall \alpha <: \tau Q_2 \text{ and } (\alpha \sqcup \tau') \in C, \tau' \notin TyVar \quad (16)$$

$$QC \implies Q(C \cup \{\tau \sqcap \tau'\}) \text{ if } (\alpha \sqcap \beta) \in C, (\forall \alpha <: \tau), (\forall \beta <: \tau') \in Q, \text{ and } Q \not\models \alpha \sqcap \beta \quad (17)$$

Fig. 4. Entailment Algorithm

To reason about correctness of the algorithm, we need to add the predicates $\tau \sqcap \tau'$ and $\tau \sqcup \tau'$ to the constraint logic with substitutions.

Definition 5 (Constraint Logic with \sqcap and \sqcup). *Extend the definition of constraint sets to include:*

$$C ::= \dots \mid \{\tau \sqcap \tau'\} \mid \{\tau \sqcup \tau'\}$$

Extend the inference rules for the subtype logic in Fig. 2 (and therefore the constraint logic rules in Def. 7) with:

$$\frac{\Gamma \vdash \tau <: \tau'' \quad \Gamma \vdash \tau' <: \tau''}{\Gamma \vdash \tau \sqcap \tau' <: \tau''} \quad (\text{SUBGLB})$$

$$\frac{\Gamma \vdash \tau'' <: \tau \quad \Gamma \vdash \tau'' <: \tau'}{\Gamma \vdash \tau \sqcup \tau' <: \tau'} \quad (\text{SUBLUB})$$

Lemma 6 (Termination). *The algorithm in Def. 3 is guaranteed to terminate.*

PROOF: Clearly this requires a loop-check; we run the algorithm until it reaches a fixed point, taking care not to redo computations. Since all constraints contain subterms of the types in the original types, and type expressions have finite height and there are a finite number of type variables, this process is guaranteed to terminate. \square

Lemma 7 (Soundness). *Suppose $\mathcal{Q}C \implies \mathcal{Q}'C'$. Then $\theta \models \mathcal{Q}C$ if and only if $\theta \models \mathcal{Q}'C'$*

PROOF SKETCH: The “if” part is trivial. For the “only if” part, for Rule (5) for example, we need to show that if $\theta \models \mathcal{Q}(\alpha <: \tau')$ then $\theta \models \mathcal{Q}(\tau <: \tau')$ (where $\theta \models \mathcal{Q}C$). We verify this by induction on the height of τ' , using the fact that uses of transitivity in the derivation of $\alpha <: \theta(\tau')$ can be pushed to the leaves of the derivation tree. Any such derivation must then be a use of SUBAX (for $\alpha <: \theta(\tau)$), and a subderivation for $\theta(\tau) <: \theta(\tau')$, followed by the use of transitivity. \square

Lemma 8 (Containment Check). *Given \mathcal{Q}, C . Then $\mathcal{Q}C$ is not satisfiable if any of the following is contained in $\mathcal{Q}C \downarrow_{<}$, but no rule of the entailment algorithm is applicable to that constraint:*

1. $(\tau <: \tau'), (\tau \sqcap \tau') \text{ or } (\tau \sqcup \tau') \in C$ where $\tau, \tau' \notin \text{TyVar}$
2. $(\alpha <: \beta) \in C$ where $\alpha, \beta \in UV(\mathcal{Q})$, and $\mathcal{Q} \not\models \alpha <: \beta$; or $(\alpha \sqcap \beta) \text{ or } (\alpha \sqcup \beta) \in C$ where $\alpha, \beta \in UV(\mathcal{Q})$, $\mathcal{Q} \not\models \alpha \sqcap \beta$; or
3. $(\tau <: \alpha), (\alpha \sqcap \tau) \text{ or } (\alpha \sqcup \tau) \in C$ where $\alpha \in UV(\mathcal{Q}), \tau \notin \text{TyVar}$; or
4. $(\alpha <: \tau) \in C$ where $\mathcal{Q} = \mathcal{Q}_1 \forall \alpha \mathcal{Q}_2$ and $\tau \notin EV(\mathcal{Q}) \cup \{\alpha\}$.

5 Finite Satisfiability

The conditions not checked for by subtype closure are (a) finiteness of solutions and (b) scoping of type variables. We are now going to define another form of closure for an

inferred constraint set, that is used to perform checks for these conditions. The details of scope checking are provided in the next section.

The subtype closure does nothing with constraints of the form $\alpha <: (\alpha * \alpha)$ (for example). In systems of recursive constraints, such constraints are satisfied by solutions involving infinite trees. Since we only allow finite solutions, the existence of such constraints in the subtype closure should flag an error. The subtype relation itself is insufficient for forming this check, consider for example:

$$\alpha <: \beta, (\alpha * \alpha) <: \beta$$

We need to define a relation that is related to subtyping, but includes symmetry. Mitchell [15] and Lincoln and Mitchell [12] use a variant of the unification algorithm to check for finite satisfiability, reflecting the fact that we need to augment subtyping with symmetry to perform the check. However they only consider subtyping between atomic types. With subtyping based on record or object interface containment, this is not sufficient. Consider these examples and their corresponding satisfying substitutions:

$$\begin{array}{ll} \{x : \alpha\} <: \beta, \alpha <: \beta & \theta(\alpha) = \{\}, \theta(\beta) = \{\} \\ \{x : \alpha\} <: \beta, \alpha <: \beta, \beta <: \{x : \gamma\} & \theta(\alpha) = \{x : \{\}\}, \theta(\beta) = \{x : \{\}\}, \theta(\gamma) = \{\} \\ \beta <: \{x : \alpha\}, \beta <: \alpha & \theta(\alpha) = \{\}, \theta(\beta) = \{x : \{\}\} \\ \{x : \alpha\} <: \alpha & \theta(\alpha) = \{\} \\ \alpha <: \{x : \beta\}, (\alpha * \alpha) <: \beta & \theta(\alpha) = \{x : \{\} * \{\}\}, \theta(\beta) = \{\} * \{\} \end{array}$$

On the other hand the constraint $(\alpha <: \{x : \alpha\})$ is not finitely satisfiable. Similarly the constraint $((\{x : \alpha\} \rightarrow \text{int}) <: \alpha)$ is not finitely satisfiable. These examples demonstrate that we need to give special treatment to circularities in the subtype constraints involving record types. We first consider the easier case of circularities not involving record types. We define a subterm relationship based on satisfying solutions for the subtype constraints:

Definition 9 (Match Closure). *Given a quantifier prefix Q and constraint formula C . Define the match closure of Q and C , denoted $QC \downarrow_M$, to be the least binary relation M such that:*

1. if $(\tau <: \tau') \in QC \downarrow_{<}$, then $(\tau \xleftrightarrow{M} \tau') \in M$;
2. $(\tau \xleftrightarrow{M} \tau) \in M$;
3. if $(\tau \xleftrightarrow{M} \tau') \in M$ then $(\tau' \xleftrightarrow{M} \tau) \in M$;
4. if $(\tau_1 \xleftrightarrow{M} \alpha), (\alpha \xleftrightarrow{M} \tau_2) \in M$, then $(\tau_1 \xleftrightarrow{M} \tau_2) \in M$;
5. if $((\tau_1 \rightarrow \tau_2) \xleftrightarrow{M} (\tau'_1 \rightarrow \tau'_2)) \in M$, then $(\tau_1 \xleftrightarrow{M} \tau'_1), (\tau_2 \xleftrightarrow{M} \tau'_2) \in M$;
6. if $((\tau_1 * \tau_2) \xleftrightarrow{M} (\tau'_1 * \tau'_2)) \in M$ then $(\tau_1 \xleftrightarrow{M} \tau'_1), (\tau_2 \xleftrightarrow{M} \tau'_2) \in M$;

It will be noted that the definition of match closure does not contain the following rule for record types:

7. if $(\{\overline{l_m} : \overline{\tau_m}\} \xleftrightarrow{M} \{\overline{l'_n} : \overline{\tau'_n}\}) \in M$, then $(\tau_i \xleftrightarrow{M} \tau'_j) \in M$ for all i, j such that $l_i = l'_j$.

The reason for this should be clear, given the above discussion.

Definition 10 (Rigid Dependency). Given \mathcal{Q} and C , $M = \mathcal{Q}C \downarrow_M$. Define that $\alpha \in EV(\mathcal{Q})$ is rigid dependent on τ with path π , written $\mathcal{Q}C \vdash_R \alpha \xrightarrow{\pi} \tau$, if the path π does not involve record types and one of the following holds:

1. $(\alpha \xleftrightarrow{M} \tau) \in M$ and $\pi = \epsilon$; or
2. $(\alpha \xleftrightarrow{M} \tau') \in M$ and $\pi = \pi_1 \pi_2$ and $\tau'(\pi_1) = \beta$, for some β , and $\mathcal{Q}C \vdash_R \beta \xrightarrow{\pi_2} \tau$.

Define $\tau \longleftrightarrow \tau$ to be: there exists a sequence of ground types τ_0, \dots, τ_n , and $\{R_1, \dots, R_n\} \subseteq \{<, >\}$, such that $\tau = \tau_1$, $\tau' = \tau_{n+1}$, and $\tau_i R_i \tau_{i+1}$ for $i = 1, \dots, n$.

Lemma 11. $\mathcal{Q}C \vdash_R \alpha \xrightarrow{\pi} \tau$ if and only if for all θ such that $\theta \models \mathcal{Q}C$, $\pi \in \text{dom}(\theta)$ and $\theta(\alpha)(\pi) \longleftrightarrow \theta(\tau)$.

For reasoning about circularities involving record types, we need a stronger condition on the occurrences of variables.

Definition 12. A path π is positive if the number of d symbols in the path is zero or even. The path is negative otherwise.

Definition 13 (Type Dependency). Given \mathcal{Q} and C . Define that $\alpha \in EV(\mathcal{Q})$ is type dependent on τ with path π , written $\mathcal{Q} \vdash_T \alpha \xrightarrow{\pi} \tau$, if there is a sequence of triples:

$$(\alpha_1, \pi_1, \tau_1), \dots, (\alpha_n, \pi_n, \tau_n)$$

where $\alpha = \alpha_1$ and $\tau = \tau_n$, and for $i = 1, \dots, n$ either:

1. $(\alpha_i <: \tau_i) \in \mathcal{Q}C \downarrow_{<}$ and π_i is a positive path, or
2. $(\tau_i <: \alpha_i) \in \mathcal{Q}C \downarrow_{<}$ and π_i is a negative path,

and furthermore $\tau_i(\pi_i) = \alpha_{i+1}$ for $i = 1, \dots, n-1$, and $\pi = \pi_1 \dots \pi_n$.

Lemma 14. $\mathcal{Q}C \vdash_T \alpha \xrightarrow{\pi} \tau$ if and only if for all θ such that $\theta \models \mathcal{Q}C$, $\pi \in \text{dom}(\theta)$ and $\theta(\alpha)(\pi) <: \theta(\tau)$.

Define $\mathcal{Q}C \vdash \alpha \xrightarrow{\pi} \tau$ to be: $\mathcal{Q}C \vdash_R \alpha \xrightarrow{\pi} \tau$ or $\mathcal{Q}C \vdash_T \alpha \xrightarrow{\pi} \tau$

Lemma 15 (Occurs Check). If $\mathcal{Q}C \vdash \alpha \xrightarrow{\pi} \alpha$ where $\pi \neq \epsilon$, then there is no finite substitution θ such that $\theta \models \mathcal{Q}C$.

Although our mini-language does not allow existential variables in the upper bounds of universal variables, this may be useful in some contexts [5]. If this is allowed, the occurs check needs to be strengthened. *Extended rigid dependency* is defined similarly to rigid dependency, with the additional clause:

3. $(\alpha \xleftrightarrow{M} \beta) \in M$, $\mathcal{Q} = \mathcal{Q}_1. \forall \beta <: \tau'. \mathcal{Q}_2$ and $\pi = \pi_1 \pi_2$ and $\tau'(\pi_1) = \gamma$ for some γ and $\mathcal{Q}C \vdash_R \gamma \xrightarrow{\pi_2} \tau$.

Consider for example $\exists \alpha. \forall \beta <: (\alpha * \alpha). \exists \gamma. \{\alpha <: \gamma, \beta <: \gamma\}$. To see why finiteness cannot be checked before entailment, consider the following constraints:

$$\exists \beta. \forall \alpha <: (\{x : \beta\} \rightarrow \text{int}). \exists \gamma. \{\alpha <: (\{x : \gamma\} \rightarrow \text{int}), \beta <: (\gamma * \gamma)\}$$

6 Scoping of Type Variables

In this section we consider the checking of scoping of type variables in the accumulated constraints. Consider the following example:

```
letmono f = λxβ.
  let g : (∀γ.γ → γ) = λy. (if true then x else y; y)
  in (if true then x else 3; g 3; g true)
in f
```

The inner conditional requires that x and y have a common upper bound. Since γ , the type of y , ranges over all possible types, this bound must be γ , so β must be instantiable to γ . But then g cannot be polymorphic in γ . In type-checking, this error is caught by the fact that β is outside the scope of γ . “Scope” is formalized by the use of a quantifier prefix, where β is introduced to the left of γ in the prefix.

More precisely type inference builds the quantifier prefix and constraint set:

$$\exists\beta.\forall\gamma.\exists\delta.\{\beta <: \delta, \gamma <: \delta\}$$

Any satisfying substitution for these constraints must instantiate β to γ , which is impossible since γ is introduced after β in the quantifier prefix. This dependency is detected by the match dependency: $QC \vdash \beta \xrightarrow{\epsilon} \gamma$.

Now consider the example:

```
let f : (∀α.α → α) = λx.
  letmono g = λyβ.
    let h : (∀γ <: α.γ → γ) = λz.
      (if true then y else z; z)
    in y
  in x
in f
```

Apparently there is another scope violation (between β and γ in this example). However γ has upper bound α , and β is within the scope of α . Therefore type subsumption can be used to replace γ as a lower bound with α , effectively moving its scope out so that more existential variables are included in its scope. More formally we have the quantified constraint set:

$$\forall\alpha.\exists\beta.\forall\gamma <: \alpha.\{\beta <: \delta, \gamma <: \delta\}$$

Since γ occurs positively in a lower bound, we can weaken the second constraint:

$$\forall\alpha.\exists\beta.\forall\gamma <: \alpha.\exists\delta.\{\beta <: \delta, \alpha <: \delta\}$$

where this step is justified by the fact that $\gamma <: \delta$ is derivable from $\gamma <: \alpha$ and $\alpha <: \delta$.

Note that this transformation is not possible if we replace the original constraint set with:

$$\forall\alpha.\exists\beta.\forall\gamma <: \alpha.\exists\delta.\{\beta <: \delta, (\gamma \rightarrow \gamma) <: \delta\}$$

In this example, the negative occurrence of γ in a lower bound prevents the above transformation.

For the case of unification under a mixed prefix, if an existential variable α may contain a variable β in its instantiation, the scope of the variable β is “moved out” (if necessary) so that the instantiation of α does not violate its scoping. For example the constraint set:

$$\exists\alpha.\forall\gamma.\exists\beta.\{\beta = \alpha, \beta = \gamma\}$$

is transformed to:

$$\exists\alpha.\exists\beta.\forall\gamma.\{\beta = \alpha, \beta = \gamma\}$$

At this point, the second constraint violates scoping since β 's scope has moved out to the scope of α .

This movement of quantifiers cannot be done with subtyping and bounds on quantified type variables. Consider for example:

$$\exists\alpha.\forall\gamma <: \tau.\exists\beta.\{\beta <: \alpha, \beta <: \gamma\}$$

Then $\{\gamma/\beta, \tau/\alpha\}$ is a satisfying substitution. This would not be a satisfying substitution if the scope of β were moved out to that of α .

Corollary 16 (Scope Check). *QC is not satisfiable if there exist α and β such that either:*

1. $Q = Q_1.\exists\alpha.Q_2.\forall\beta <: \tau.Q_3$ (or $Q = Q_1.\exists\alpha.Q_2.\forall\beta.Q_3$) and α is type dependent on β ; or
2. $Q = Q_1.\exists\alpha.Q_2.\forall\beta.Q_3$ and α is rigid dependent on β .

The verification relies on Lemma 11 and Lemma 14 from the previous section. There we considered “false circularities” due to existential type variables in record field types that could be omitted through record type subsumption. Here we consider “false scope violations” due to universal type variables that can be omitted by subsuming them with their upper bounds.

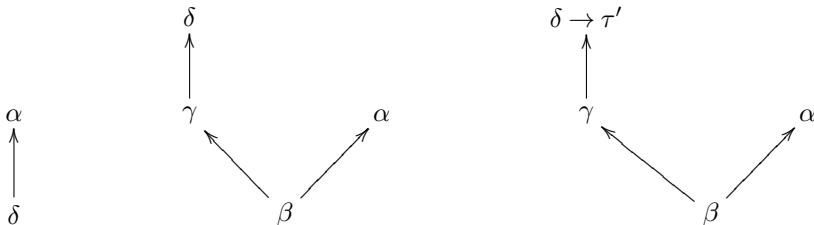
If the scope check does not fail, the subtype constraints are well scoped. The following examples illustrate the rationale for this. Consider the quantified constraint sets:

$$\exists\alpha.\forall\delta <: \tau.\{\delta <: \alpha\}$$

$$\exists\alpha.\forall\delta <: \tau.\exists\beta.\exists\gamma.\{\beta <: \alpha, \beta <: \gamma, \gamma <: \delta\}$$

$$\exists\alpha.\forall\delta <: \tau.\exists\beta.\exists\gamma.\{\beta <: \alpha, \beta <: \gamma, \gamma <: \delta \rightarrow \tau'\}$$

All of these are examples where $QC \vdash_R \alpha \xrightarrow{\epsilon} \delta$, but where the dependency is not a type dependency. These examples are represented graphically as:



The first of these corresponds to a positive occurrence of δ in a lower bound for α . This constraint can be instantiated by instantiating α to τ . This is not possible if δ occurs positively in an upper bound of α . The second example corresponds to $QC \vdash \alpha \xrightarrow{\epsilon} \delta$, where the dependency path is through a path of subtype and supertype edges. The positive occurrence of δ in upper bounds for γ and β requires that these variables be instantiated to δ in any satisfying substitution. However because the dependency of α on δ relies on the lower bound constraint ($\beta <: \alpha$), it is possible to instantiate α to τ . The third example demonstrates a negative occurrence of δ in the upper bound for γ and β . In this case it is possible to instantiate all of α , β and γ to $\tau \rightarrow \tau'$.

7 Correctness of Type Inference

We finally consider the correctness of the type inference algorithm.

Lemma 17. *If QC does not violate the entailment check, occurs check or scope check, then $\models QC$.*

PROOF SKETCH: The proof is constructive, instantiating existential variables to their least upper bounds. Where a variable α has a dependency on a universal variable β to the right of it in the quantifier prefix, then the instantiation of α includes some upper bound of β that is in scope. This is possible because the dependency is not hard. \square

Lemma 18. *Given an execution of the type inference algorithm, and an application of LET in this execution that introduces universal variables $\{\bar{\alpha}\}$, with outputs Q^{local} and C^{local} , while type-checking the definition in a let. Let Q^{global} and C^{global} be the global outputs of the algorithm. Then for any $\beta \in EV(Q^{\text{global}})$, $\alpha \in \{\bar{\alpha}\}$, we have $Q^{\text{local}}C^{\text{local}} \vdash \beta \xrightarrow{\pi} \alpha$ if and only if $Q^{\text{global}}C^{\text{global}} \vdash \beta \xrightarrow{\pi} \alpha$.*

PROOF: Suppose $Q^{\text{local}}C^{\text{local}} \not\vdash \beta \xrightarrow{\pi} \alpha$ and $Q^{\text{global}}C^{\text{global}} \vdash \beta \xrightarrow{\pi} \alpha$. This must be due to the addition of a subtype constraint that relates β to α or to some γ such that $C^{\text{local}} \vdash \gamma \xrightarrow{\pi} \alpha$. The former case cannot happen because $\{\bar{\alpha}\}$ are not free in the type environment outside the scope of the let. In the latter case, γ must be free in the environment outside the scope of the let. But since $Q^{\text{local}}C^{\text{local}} \vdash \gamma \xrightarrow{\pi} \alpha$, γ must also be free in the environment while type-checking the let. Therefore γ must be to the left of α in the quantifier prefix (since γ was added to the environment before the let was type-checked), which violates the scope check. \square

We verify the following by induction on the execution of the type inference algorithm, using Lemma 17 and Lemma 18.

Theorem 19. (SOUNDNESS OF TYPE INFERENCE) *Given expression e , type environment A , and quantifier prefix Q . Suppose $Q; A \vdash e : \tau \mid Q', C$. Let θ be the satisfying substitution computed in the proof of Lemma 17. Let $\Gamma = \{\alpha <: \tau \mid (\forall \alpha <: \tau) \in Q\}$. Then $\Gamma; \theta(A) \vdash e : \theta(\tau)$.*

Theorem 20. (COMPLETENESS OF TYPE INFERENCE) *Given $\text{dom}(\theta) \subseteq EV(Q)$. Given $\Gamma; \theta(A) \vdash e : \tau$, and $Q; A \vdash e : \tau' \mid Q', C$ where $\text{dom}(\theta) \subseteq FV(A)$. Then there exists θ', θ'' such that $\theta' = \theta'' \circ \theta$ and $\theta' \models (Q'C)$ and $\Gamma \vdash \theta'(\tau') <: \tau$.*

8 Conclusions

Finite subtype inference occupies a middle ground between Hindley-Milner type inference (as in ML) and subtype inference with recursively constrained types. We have presented algorithms for finite subtype inference, including checking for entailment of inferred types against explicitly declared polymorphic types. This resolves for finite types a problem that is still open for recursively constrained types. Some motivation for this work, particularly for finite types and explicit polymorphism, is in providing subtype inference for first-class container objects with polymorphic methods.

Flanagan and Felleisen give algorithms for checking entailment of recursive set constraints arising from data-flow analysis [8]. Although superficially similar to constraint systems for object-oriented languages, there are in fact subtle but significant differences in the underlying languages, and it is not clear if their techniques can be adapted to solve the problem of entailment for recursive types.

Bourdoncle and Merz [4] provide a type-checking algorithm for an ML dialect where subtyping is declared between class types. Their work is more related to earlier work on finite atomic subtyping than the current work, and they do not have a structural notion of subtyping.

The other relevant work is that of Pierce and Turner [18] on local type inference. They work with an impredicative type system, and allow type annotations to be omitted where it is possible to infer the type from an “upper bound” type constraint. The work presented here may be seen as a more flexible approach, where type annotations are only required for polymorphic function definitions. Pierce and Turner require that it be possible (using meets and joins) to compute the specific type of every expression. Where a type variable occurs in invariant position (for example in the element type of a mutable reference cell), their algorithm may fail to determine a type. Pottier reports that type variables in invariant position are reasonably common in practical subtype inference. Although we have not elaborated upon it here, impredicativity can be incorporated using objects with polymorphic methods [5].

Acknowledgements. Thanks to Scott Smith and Benjamin Pierce for helpful discussions.

References

1. Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, July 1996.
2. A. Aiken and E. Wimmers. Solving systems of set constraints. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 329–340, June 1992.
3. A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of ACM Symposium on Functional Programming and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.
4. Francois Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1997.
5. Dominic Duggan. Object type constructors. In preparation. A preliminary version appeared in the *Workshop on Foundations of Object-Oriented Languages*, New Brunswick, New Jersey, July 1996.

6. Dominic Duggan. Unification with extended patterns. *Theoretical Computer Science*, 1997. To appear.
7. Jonathan Eifrig, Scott Smith, and V. Trifonov. Type inference for recursively constrained types and its application to oop. In *Proceedings of the Conference on Mathematical Foundations of Programming Semantics*, Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.
8. Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.
9. You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73:155–175, 1990.
10. Mark Jones. First-class polymorphism with type inference. In *Proceedings of ACM Symposium on Principles of Programming Languages*. ACM Press, January 1997.
11. Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.
12. Patrick Lincoln and John C. Mitchell. Algorithmic aspects of type inference with subtypes. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 293–304. ACM Press, 1992.
13. Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.
14. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17:343–378, 1978.
15. John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–286, July 1991.
16. Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1996.
17. Jens Palsberg. Efficient inference of object types. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 186–195, Paris, France, July 1994. IEEE.
18. Benjamin Pierce and David Turner. Local type inference. In *Proceedings of ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1998. ACM Press.
19. Francois Pottier. Simplifying subtyping constraints. In *Proceedings of ACM International Conference on Functional Programming*, Philadelphia, Pennsylvania, May 1996. ACM Press.
20. Didier Rémy. Programming objects with ML-ART: An extension to ml with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.
21. Marvin Solomon. Type definitions with parameters. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 31–38. ACM Press, 1978.
22. Valery Trifonov and Scott Smith. Subtyping constrained types. In *Static Analysis Symposium*, number 1145 in *lncs*, pages 349–365, 1996.

Sparse Jacobian Computation in Automatic Differentiation by Static Program Analysis

M. Tadjouddine, F. Eyssette, and C. Faure

Projet Safir, INRIA Sophia Antipolis
e-mail: {tadjouddine, faure, eyssette}@sophia.inria.fr

Abstract. A major difficulty in quickly computing Jacobians by Automatic Differentiation is to deal with the nonzero structures of sparse matrices. We propose to detect the sparsity structure of Jacobians by static program analysis. The method consists in traversing the data dependence graph extended with the control-flow of the program and computing relations between array regions. Then, we safely extract informations about the dependences from program inputs to program outputs. The generation of the derived program uses these informations to produce a better result. We eventually, introduce the Automatic Differentiation tool *Odyssée* and present some benchmark tests.

1 Introduction

The computation of derivatives plays an important role in areas like inverse problems, optimum design, and nonlinear modeling. Automatic Differentiation is a solution to get fast and accurate derivatives of a function represented as a program. To perform Automatic Differentiation, one of the used mechanisms is the source-to-source translation code. In most cases, the derived code for Jacobian computation is sparse. A disadvantage is that the generated code spends most of the time computing and storing zero elements. An other disadvantage is that sparse computations lead irregular data accesses that can prohibit the use of the cache and then reduce memory bandwidth. Those zero elements can be induced either by the inputs of the program or the dependences between cells of the array variables of the program. This observation leads to a strategy for detecting and exploiting the nonzero structures of the Jacobians.

To solve this problem, one can distinguish at least two main approaches. The first one is the dynamic approach that consists in computing the set of nonzero elements at run-time. We have studied the second one which is the static approach. This approach is based on the a priori detection of the set of nonzero elements by the localization of dependences between array elements.

The dynamic approach has been studied, and several different methods have been tested. General techniques related to node elimination [24], graph reordering [24], and graph coloring [10] are known but they rely on a priori knowledge of the sparsity structure. In the context of sparse Jacobian computation by Automatic Differentiation, Griewank and Reese [16] have proposed to use Markowitz

Elimination [24]. In this method, the execution of a program is viewed as a directed acyclic graph. The nonzero inputs of the program are directly connected to the outputs of the program by eliminating intermediate vertices. Geitner and al. [17] have extended the method of Newsam and Ramsdell [23]. The basic idea is to find a seed matrix S such that a Jacobian J can be optimally reconstructed from the product JS . These two methods have been tested in **Adol-C** [15]. Bischof and al. [3] have interfaced **Adifor** [3] with the **SparseLinC** [3] library. The structure of the Jacobian may be investigated with the graph coloring techniques [10]. Although these techniques improve the efficiency of Automatic Differentiation, they are costly.

The only static technique (see Bischof and al. [3]; Rostaing and al. [26]) applied in Automatic Differentiation is based on dependence analysis, which treats arrays as atoms. This approach cannot detect the sparsity structure of a Jacobian. In our approach, we statically analyze the dependences between array regions relying on experience from compilers and parallelizers. In the context of sparse program analysis, Bik and Wijshoff [5] proposed to improve the performance of compiler by computing the Data Access Descriptor of arrays with the traversal order, and subsequent a dynamic compact data structure is built for a translation of the source code using a library of sparse primitives. The Data Access Descriptor has been introduced by Balasundaram and Kennedy in [6]. To increase the automatic parallelization of sparse programs, Kessler [20] developed a library of generic patterns. The method consists in recognizing patterns in the source code by cross-matching using knowledge of this library of templates.

Our aim is to detect the sparsity structure by dependence analysis of array regions and to exploit this sparsity to generate more efficient derivative code. Because the static analyses cannot be exact in some cases, we use conservative approximations. The choice of approximations is determined by a trade-off between complexity and precision. This choice is based on the patterns that occur frequently in actual programs and the typical information that may be used efficiently in our particular application, Automatic Differentiation.

The sparse Jacobian computation proposed in this paper consists of three phases. In the first phase, we safely extract dependency informations from the original program. The corresponding algorithm consists in doing a computation on a subgraph of the data dependence graph extended with the control-flow (CDFG). That computation yields access regions of arrays and relations between dependent regions. In the second phase, we present an automatic data structure selection followed by a resolution of possible conflicts. In the third phase, we generate the derivative code using the informations computed in the first two phases.

We implemented this method for single Fortran unit by using two tools developed at Inria: **Partita** [18], a tool for Automatic Parallelization of Fortran codes and **Odyssee** [26], [12] an Automatic Differentiation tool. **Partita** provides the CDFG of the program. Using this CDFG, we compute the Differentiation Dependence Graph (DDG) of the program. This DDG is a bipartite graph containing dependence informations for Automatic Differentiation. Then, we send

this DDG to the **Odyssée** tool that uses these informations for computing the derivative of the source program.

The remainder of this paper is organized as follows: Section 2 presents a brief review of Automatic Differentiation. Section 3 describes the detection of the sparse Jacobian structure. Section 4 discusses the selection of data structure for sparse derivatives. Section 5 shows the way the dependency information is used for Jacobian code generation. Section 6 presents the **Odyssée** tool and some experimental results. Finally, we present our conclusions in Sect. 7.

2 Automatic Differentiation (AD)

Automatic Differentiation (AD) is a set of techniques for computing derivatives from any already written code. Automatic Differentiation is based on two fundamental principles : any program can be seen as a composition of elementary functions and can then be differentiated using the chain rule.

With this technique, the generated codes compute derivatives which are as precise as the initial code and with a well known complexity. With software tools such as **Odyssée** [26; 12], **Adifor** [3], **Adol-C** [15]; and **Tamc** [14] AD is gaining popularity in industrial applications. An other way of getting derivative from a code is using finite differences (see [13]), it is easy to implement but leads to numerical errors and the time complexity grows with the number of the variables [3]. Also, one can compute derivatives by differentiating the program by hand. This last approach is tedious and error prone but the but somehow optimal.

In fact, AD of programs consists in creating extra statements that compute the derivatives of the original statements. Thereby we automatically construct a new program that computes values and derivatives of the function represented by the initial program. On that account, the classical formulas for computing derivatives of functions are used. For instance, denoting dv the directional derivative of any variable v , we get:

$$\begin{aligned} s &= x + y \Rightarrow ds = dx + dy \\ s &= x * y \Rightarrow ds = dx * y + x * dy \\ s &= f(x, y) \Rightarrow ds = \frac{\partial f}{\partial x} * dx + \frac{\partial f}{\partial y} * dy \end{aligned}$$

Figure 1 illustrates a AD process for computing directional derivatives. A Jacobian can be obtained by computing all the directional derivatives. We construct the Jacobian code by creating the partial derivatives as follows. If x and y are intermediate variables that depend on a program input variable z , then

$$s = f(x, y) \Rightarrow \frac{\partial s}{\partial z} = \frac{\partial f}{\partial x} * \frac{\partial x}{\partial z} + \frac{\partial f}{\partial y} * \frac{\partial y}{\partial z}$$

There are 2 modes of Automatic Differentiation, both having predictable complexity (see [13]):

<pre> s = 0. do i=1, 10 s = s+x(i)*y(i) end do s = sqrt(s) </pre>	→	<pre> ds = 0. s = 0. do i=1, 10 ds = ds+dx(i)*y(i)+x(i)*dy(i) s = s+x(i)*y(i) end do ds = ds/2.*sqrt(s) s = sqrt(s) </pre>
---	---	--

Fig. 1. An example of AD

1. The forward mode in which the intermediate derivatives are computed in the same order as the program computes the composition.
2. The reverse mode in which the intermediate derivatives are computed in the reverse order. The reverse mode is optimal for computing gradients because its complexity is independent from the number of the input variables [13].

To implement the AD process, we distinguish two main approaches:

- Operator overloading consists in overloading the atomic operations (basic arithmetic operators, intrinsic calls) to hold the propagation of derivatives. For instance, *Adol-C* [15] uses this mechanism.
- Source-to-source transformation relies on compiler techniques. Actually, a source code is transformed to a new source code by insertion extra statements. *Adic* [4], *Adifor* [3], *Odyssée* [26], [12], and *Tamc* [14] use this mechanism.

3 Sparse Jacobian Analysis

In this section we describe the approach we have used for statically detecting sparsity structure of Jacobians. Let us illustrate our motivation on a simple example.

Let us consider the following piece of code:

```

do i=1,n
  a(i) = b(2*i)
enddo

```

Assume we want to compute $\frac{\partial a}{\partial b}$ denoted $dadb$ where the array b gets directly its input value. If one just considers that a depends on b , one must generate the following piece of code in which lb and ub are the lower and upper bound of the array b :

```

do i=1,n
  do j=lb, ub
    dadb(i,j) = dbdb(2*i,j)
  end do
  a(i) = b(2*i)
enddo

```

On the other hand, if it is known that $a(i)$ depends only on $b(2*i)$, more efficient piece of code can be generated:

```
do i=1,n
  dadb(i) = dbdb(2*i)
  a(i) = b(2*i)
enddo
```

In the first case, $\frac{\partial a}{\partial b}$ is a sparse two-dimensional array. This is caused by the fact that the information ($\forall i \in [1 : n]$ $a(i)$ depends only on $b(2*i)$) implies $dadb(i, j) = 0$ if $j \neq 2*i$. In the second case, $\frac{\partial a}{\partial b}$ is computed as a diagonal vector using this dependence information. We notice the following observations:

Observation 1: Instances of statements in which the derivative value is zero can be eliminated.

Observation 2: A value of a derivative $\frac{\partial a}{\partial b}_{ij}$ at point (i, j) is zero when $a(i)$ does not depend on $b(j)$. In other words, for given i and j , we get:

$$\frac{\partial a}{\partial b}_{ij} \neq 0 \Rightarrow a(i) \text{ depends on } b(j).$$

3.1 Problem Statement

If many elements of an $n \times m$ matrix A are zero, then A is called *sparse* matrix. Otherwise, it is *dense*. When a matrix is sparse, it seems worthwhile to reduce the storage requirement by storing as long as possible only the nonzero elements.

Traditionally, data dependence is defined as a relation between two statements that expresses that both access one or more common memory locations [28]. Automatic Differentiation operates on the values of the variables of the program. That implies the order in which these values are executed. For that reason, we define a differentiation dependence notion as follows:

Definition 1. Let v_1 and v_2 be two memory references in a program. Then, there is a differentiation dependence from v_1 to v_2 if v_2 uses v_1 during its calculation. We denote this relation: $v_2 \delta v_1$.

Let us now establish the connection with the classical dependences [28]. We say that $v_2 \delta v_1$ if and only if there is a **flow dependence** or a **value dependence** from v_2 to v_1 . As described in [18] we recall that **the flow dependence** is from a Write atomic operation of a memory location to a Read atomic operation that may access the value written at this same location and **the value dependence** is from an atomic operation returning a value on the execution stack to the atomic operation using this value.

Let a and b two arrays of size n and m respectively. Let $D(a, b)$ be the set of differentiation dependences between the cells of a and the cells of b , $D(a, b)$ is described by the relation:

$$\{i \rightarrow j \mid 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge a(i) \delta b(j)\}.$$

Let NZ be the set of non zero elements of the matrix $\frac{\partial a}{\partial b}$. One can easily point out that:

$$NZ \subseteq D(a, b) \subseteq [1 : n] \times [1 : m].$$

This inequality implies that NZ can be approximated by $D(a, b)$. Our purpose is to compute the set $D(a, b)$ for any couple of arrays a and b of the program. For that reason, we use an analysis of array regions that consists in computing access regions and relations between dependent regions, see [25].

3.2 Array Regions Analysis

Several representations of array regions have been proposed [6], [19], [22], or [11]. Some of them are more precise but expensive in terms of complexity. We have observed that in most programs to be differentiated, access patterns of arrays and bounds of the loops are affine. Moreover, the generated code has to be efficient and readable by human beings. We have chosen **simple regions** as an appropriate representation of the index set of an access part in an array.

Definition 2. *A simple region is a Cartesian product of regular intervals. If S is a simple region, it is denoted by $S = \times_{j=1}^n [l_j : u_j : p_j]$, where $[l_j : u_j : p_j]$ is the integer set going from l_j to u_j by increments of p_j .*

Because of undecidability or high complexity, an array region must be safely approximated. Ad-hoc solutions derived from abstract interpretation [7] have been proposed. The approximate regions are computed as least fixpoints by successive iterations based on sequences of *widening/narrowing* operators. The trapezoidal congruences [22] for summarizing array accesses are designed using abstract interpretation. We compute exact regions of arrays and we switch to over-approximation when necessary. This solution is discussed in [11].

Obviously, the simple region needs over-approximations. In other words, in addition to the accessed points, the simple region can include non accessed points. Generally, over-approximate analyses are defined on lattices, for instance lattice of convex polyhedra in [11] or Regular Section Descriptor (RSD) in [9]. The simple region used in this paper is similar to the bounded regular sections [19] that is an example of RSD. Similarly, the semantic of over-approximate analysis by simple regions is well defined on the lattice of simple regions.

Moreover, affine functions are not the only subscript form of the array occurrences appearing in a program. We over-approximate all other access patterns of array by the access domain authorized by the program declaration.

Operators on Simple Regions. A dependence between array regions is the composition of the memory effects on array elements at each step of the propagation across the CDFG. In order to propagate this information, we define Intersection, Union, Difference over simple regions of arrays.

These operators are computed as exact or over-approximate. For more details on approximations of these operators, see [8], [11]. Intersection enables to test the dependence between two regions of the same array. Union allows to merge

two access regions of an array in two points of the program. Difference is used to obtain the index set of the array that are not killed at a step of the propagation.

Regions of Arrays. We have chosen to handle formally the operators (Union, Intersection, Difference) on simple regions with the possibility of using a rough over-approximation as soon as the number of the components of the present expression exceeds a fixed limit. We use the following definition to describe a region R of a given array:

$$R ::= \text{Simple region} \mid R \cap R \mid R \cup R \mid R \setminus R$$

These expressions are evaluated for integer bounds and simplified for symbolic bounds.

3.3 A Descriptor for Trapezoidal Matrices

Let a, b be two arrays, R_a and R_b be the access regions of a and b respectively. We denote Ψ the function from R_a to R_b defined as follows : $\forall i \in R_a, \Psi(i) = \{j \in R_b \mid a(i) \delta b(j)\}$. We summarize $D(a, b)$ (the set of differentiation dependences between two arrays a and b) by a list of triplet $\langle R_a, R_b, \Psi \rangle$.

For instance, in Fig. 2, if $\wp([0 : 10 : 1])$ represents the set of parts of the integer set $[0 : 10 : 1]$, then the matrix $dadb$ will be represented by $\langle [2 : 7 : 1], [0 : 10 : 1], \Psi \rangle$ with Ψ defined as follows:

$$\begin{aligned} \Psi : [2 : 7 : 1] &\rightarrow \wp([0 : 10 : 1]) \\ i &\mapsto \{2, 2i - 4, 8 - i\} \end{aligned}$$

This descriptor enables us to represent a collection of diagonal vectors, anti-diagonal vectors or column vectors, triangular matrix, or generally trapezoidal matrix.

We distinguish the following data type for the storage scheme of sparse Jacobian:

- DIAG for representing the form consisting of vectors collection.
- TRAP for a triangular form or generally, a trapezoidal form.
- ALL for all forms not detected by our analysis.

3.4 Computing the Differentiation Dependence Graph (DDG)

We call Differentiation Dependence Graph (DDG) of a program, a bipartite graph in which each Write variable is related to all the Inputs of the program that impact it. The Control Flow Graph (CFG) plays a central role in compilation (code optimization). But the propagation of informations across the CFG requires iterative methods [1]. In some cases, this propagation is costly and not

```

do i =2, 7
  a(i) = b(2)+b(2*i-4)+b(8-i)
end do

```

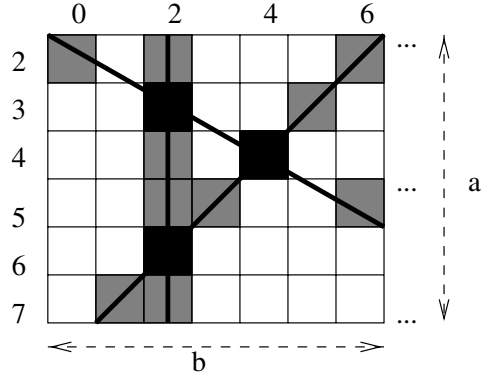


Fig. 2. Representation by a set of vectors

necessarily precise because of the number of iterations needed by the convergence of the method. We have chosen to use the CDFG because it allows us to examine directly an output of the program as a function of the inputs of the program.

The method used for computing the DDG consists in doing a depth search from each output of the program (dependent variable) until the program inputs (independent variables). We use Tarjan's algorithm [27] for computing the strongly connected components of graph. Each component is treated in order to find the dependences relating the Write variables to the Input variables of the program.

Traversing the CDFG. The CDFG of Partita [18] is a directed graph stored as a list of nodes where each node contains the list of arrows related to it. A node represents an atomic operation: Read/Write of a variable, an evaluation of arithmetic operations (+, *, ...), an evaluation of boolean operation, a call of function, or a loop counter. An arrow represents a dependence between two atomic operations. According to our definition of differentiation dependence, only arrows that represent a flow or value dependences are traversed. We call Value Dependence Graph (VDG) the subgraph consisted of these two types of dependence.

We define a partial order in the VDG according to the Fortran semantic and we do a depth search of this VDG. The dependence $d(x)$ at a node x is the union of all dependences computed on all execution paths coming to x . If $P(x)$ represents the set of predecessors of x , then

$$d(x) = \cup_{y \in P(x)} d(y).$$

Between two occurrences of arrays that are actually nodes of the VDG, we compute a relation that associates an index tuple of an array to an other. We compose these relations while traversing the graph. We obtain tuple relations by

computing transitive closure, see [21]. The resulting relation maps elements of two dependent regions, a region of Write variable and a region of Input variable of the program.

Summary Differentiation Dependences. To describe the dependences of a program variable v in a given region, we simply keep the list of the variable references whose v depends on. However, a same variable can be calculated in two different points S_1 and S_2 of the control flow of the program. When S_2 is always executed after the execution of S_1 , we say S_2 succeeds S_1 and we note it $S_2 \gg S_1$.

To illustrate this summary, let us consider the following situation:

$$\begin{aligned} S_1 : \forall i \in R_1, \quad a(i) \text{ depends on } x \\ S_2 : \forall i \in R_2, \quad a(i) \text{ depends on } y \end{aligned}$$

- If $\langle S_2 \gg S_1 \rangle$, then
 - $\forall i \in R_1 \setminus R_2$, $a(i)$ depends on x .
 - $\forall i \in R_2$, $a(i)$ depends on y .
- Otherwise, $\forall i \in R_1 \cup R_2$, $a(i)$ depends on x and y .

If $i \in R_1 \cap R_2$, we get $a(i)$ depends on x . Since $S_2 \gg S_1$, the preceding dependency is cancelled, and thus $a(i)$ depends only on y . If $i \in R_1 \cup R_2$, the system cannot determine what dependency is killed and computes an over-approximation by the union of dependences.

4 Data Structure Selection and Conflicts Resolving

To derive a code by Automatic Differentiation, one can identify the statements where the derived code will be sparse. Then a compact data structure must be selected for each matrix. In order to generate valid derivative code, we must ensure the consistency of the representations of sparse matrices by resolving the possible conflicts.

4.1 Choice of Data Structure

We describe the set of differentiation dependences of a derivative matrix as a dense region with the help of the descriptor of trapezoidal matrices, see Sect. 3. Hence, for the computation of a derivative, a dense storage is chosen automatically. Direction vectors are used for computing the elements of the descriptor, see [5]. For instance, a tridiagonal matrix will be stored as follows:

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{pmatrix} \rightarrow \begin{array}{|c|c|c|} \hline a_{21} & a_{11} & - \\ \hline a_{32} & a_{22} & a_{12} \\ \hline a_{43} & a_{33} & a_{23} \\ \hline - & a_{44} & a_{34} \\ \hline \end{array}$$

For each $n \times m$ derivative matrix $\frac{\partial a}{\partial b}$ whose structure is described by $D(a, b)$, an abstract representation $AR(a, b)$ is built by using a bijective function $addr_{a,b}$ described as follows:

$$\begin{aligned} addr_{a,b} : D(a, b) &\rightarrow AR(a, b) \\ (i, j) &\mapsto addr_{a,b}(i, j) \end{aligned}$$

The function $addr_{a,b}$ maps each element (i, j) of the set $D(a, b)$ onto an address $addr_{a,b}(i, j)$ of the address set $AR(a, b)$. Because $addr_{a,b}$ is a bijective function, we preserve precise informations on the set of indices for each calculated sparse derivative. In our case, the part of the derivative matrix that must be calculated is fixed at compile-time. However, at run-time one must adapt the function $addr_{a,b}$ and the set $D(a, b)$ to account for the insertion of a new element or the cancellation of an old element.

In order to generate efficient code by Automatic Differentiation, we need to construct an abstract representation $AR(a, b)$ such that:

- An address of a valid element of the set $D(a, b)$ is easy to generate.
- A value $\frac{\partial a}{\partial b}_{ij}$ of the matrix $\frac{\partial a}{\partial b}$ is stored once.

Moreover, the description of the nonzero structure $D(a, b)$ of a derivative matrix $\frac{\partial a}{\partial b}$ is based on the summary of regions accessed along all the possible paths of the control-flow of the program. Conflicts arise when at least two components of $D(a, b)$ intersect or overlap. To generate valid derivatives, one must resolve these conflicts.

4.2 Conflicts Resolving

To solve the possible conflicts between components of the set $D(a, b)$, we must construct a summary set $S_{(a,b)} = \{D_1, D_2, \dots, D_s\}$ such that:

1. $\forall i, j \in [1 : s], i \neq j, D_i \cap D_j = \emptyset$.
2. $\cup_{i=1}^s D_i = D(a, b)$.

The summary set $S_{(a,b)}$ is useful because the elements of an intersection of components of the set $D(a, b)$ needs particular treatments. For instance, assume the differentiation dependence $a(i) \delta b(j)$ is checked at a point of the program. If this dependency is removed at an other point of the program, then the derivative $\frac{\partial a(i)}{\partial b(j)}$ becomes zero. On the other hand, assume a derivative matrix $\frac{\partial a}{\partial b}$ is described by the following set $D(a, b)$ in which f_1 and f_2 are integer functions:

$$\begin{aligned} D(a, b) = & \{(i, j) \in \mathbb{Z}^2 \mid 1 \leq i \leq n \wedge j = f_1(i)\} \\ & \cup \{(i, j) \in \mathbb{Z}^2 \mid 1 \leq i \leq n \wedge j = f_2(i)\}. \end{aligned}$$

Let $i_0 \in [1 : n]$ such that $f_1(i_0) = f_2(i_0) = j_0$. If in the original code, we have $a(i) = b(f_1(i)) + b(f_2(i))$, then the derivative at the point (i_0, j_0) is obtained as follows: $\frac{\partial a(i_0)}{\partial b(j_0)} = \frac{\partial b(f_1(i_0))}{\partial b(j_0)} + \frac{\partial b(f_2(i_0))}{\partial b(j_0)}$. In other cases, $\frac{\partial a(i)}{\partial b(j)} = \frac{\partial b(f_1(i))}{\partial b(j)}$ or $\frac{\partial b(f_2(i))}{\partial b(j)}$.

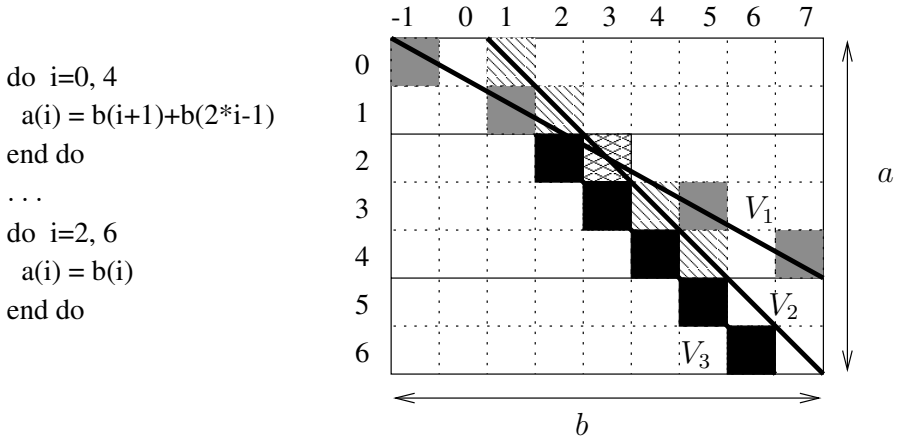


Fig. 3. Conflicts between V_1 , V_2 , and V_3

To illustrate the two situations above, let us consider the example shown in Fig. 3.

Figure 3 shows that the derivative $\frac{\partial a}{\partial b}$ can be stored and computed as three vectors V_1 , V_2 and V_3 . At the point of intersection of V_1 and V_2 , the value of the derivative is the sum of the two values calculated in the directions of V_1 and V_2 . Because $[0 : 4]$ and $[2 : 6]$ overlap in $[2 : 4] \times [2 : 7]$, some differentiation dependences between cells of a and b may remove the differentiation dependences already determined in the first loop. In this case, an element of V_1 or V_2 belonging to $[2 : 4] \times [2 : 7]$ becomes zero and must be cancelled. In other words, values of V_3 can kill some values of V_1 or V_2 . The question is how to generate a valid derivative with these conflicts.

Introducing Conditions. A way of resolving these conflicts consists in using conditionals. For each component of the set $D(a, b)$ of a matrix $\frac{\partial a}{\partial b}$, a basis composed of directional vectors is computed. Then we test if this component may intersect an other one. We reduce the use of If-statements by testing in advance whether the vectors of basis are linearly independent. For instance, since the two hyperplans described by the equations $j = 2i - 1$ and $j = 2i + 3$ are parallel, it is useless to test if $2i - 1 = 2i + 3$. A derivative of the code in Fig. 3 can be computed as shown in Fig. 4 (a).

However, this method is costly because the number of tests which may be performed is proportional to the number of iterations of the loop.

Splitting of Loops. An other way of overcoming these conflicts is to use index set splitting as discussed in [28]. These conflicts are caused by the fact that simple regions in the the set $D(a, b)$ intersect or overlap. Hence, we can fragment

these simple regions according to their different common elements. The summary set $S_{(a,b)}$ constructed from $D(a,b)$ will be obtained as a union of disjoint components. For the example of Fig. 3, one can generate the derivative code shown in Fig. 4 (b).

This method needs an effective computation of a summary set $S_{(a,b)}$ of a derivative matrix followed by a transformation of loops. A survey of loop transformations can be found in [2].

<pre> (a) do i=0, 4 if (i+1.eq.2*i-1) then dadb1(i)=dbdb(i+1)+dbdb(2*i-1) dadb2(i)=dadb1(i) else dadb1(i)=dbdb(i+1) dadb2(i)=dbdb(2*i-1) endif a(i)=b(i+1)+b(2*i-1) enddo ... do i=2, 6 if ((i.ge.2).and.(i.le.4)) then dadb1(i)=0. dadb2(i)=0. endif dadb3(i)=dbdb(i) a(i) = b(i) enddo </pre>	<pre> (b) do i=0, 1 dadb1(i)=dbdb(i+1) dadb2(i)=dbdb(2*i-1) enddo dadb1(2)=dbdb(3)+dbdb(3) dadb2(2)=dadb1(2) do i=3, 4 dadb1(i)=dbdb(i+1) dadb2(i)=dbdb(2*i-1) enddo do i=0, 4 a(i) = b(i+1)+b(2*i-1) enddo ... do i=2, 4 dadb1(i)=0. dadb2(i)=0. enddo do i=2, 6 dadb3(i)=dbdb(i) a(i) = b(i) enddo </pre>
--	--

Fig. 4. Solutions to the problem of Fig. 3

5 Derivative Code Generation

As discussed in Sect. 3, the Sparse Jacobian analysis enables us to compute safe informations relating to the structure of each derivative variable. These informations are stored in the DDG, which stores the locality and the validity of each derivative variable.

For each statement in the initial code, we create corresponding derivative statements (see Sect. 2) by using the information stored in the DDG. Each structure of derivative is scanned by generating Do-loops or If-statements, see the examples of the previous section. Furthermore, the entire program transformation involves declarations of the new variables possibly followed by initializations.

5.1 Declarations

The phase of program analysis enables us to detect all the points of the program where a derivative must be calculated. This implies local informations. But at the end of the program, we need a global information. Hence, for each derivative $\frac{\partial a}{\partial b}$, we compute the storage summary set $S_{a,b}$ as the small simple region describing all access regions of the matrix $\frac{\partial a}{\partial b}$. If we denote $S_{a,b}$ by $\langle R_a, R_b, \Psi \rangle$, for any derivative $\frac{\partial a}{\partial b}$, then its declaration is generated as follows:

- If for all $i \in R_a$, $\Psi(i)$ is an exhaustive list of patterns, then its declaration is generated as follows:

```
...
TYPE dadb( $r, n$ )
...
```

In this declaration, TYPE is the unified type of the types of the variables a and b , r is a range $[l : u]$ where l and u are respectively the lower and upper bounds of the region R_a , and finally n represents the number of patterns stored in the list of indices $\Psi(i)$.

- If for all $i \in R_a$, $\Psi(i)$ is a simple region, then the following declaration in which r_1 and r_2 are the ranges related to the regions R_a and R_b respectively is generated:

```
...
TYPE dadb( $r_1, r_2$ )
...
```

5.2 Initializations

While deriving the code, an initialization process is used whenever it is necessary for all descriptor of a derivative matrix. It consists in generating appropriate code at the beginning of the main program. Actually, the system scans the nonzero structure which has been selected for a derivative matrix and initializes each of its elements. For a storage summary $S_{(a,b)}$ of a derivative $\frac{\partial a}{\partial b}$, one can generate for example the following code:

```
...
do  $i \in S_{(a,b)}$ 
  dadb(i) = 0.
enddo      ↑ constant of appropriate type.
...
```

However, not all the derivatives need to be initialized. Only the derivatives which are used before their calculation must be initialized. They are determined by an examination of the dependence informations stored along the control-flow graph of the program.

6 Experimental Results

As described at the end of the section 1, we compute the DDG under **Partita** and send it to **Odyssée**. **Odyssée** generates the code by using the informations stored in this DDG.

6.1 Odyssée

In a few words, **Odyssée** is a tool-box for Automatic Differentiation of **Fortran** 77 programs. It performs a source-to-source transformation of a program by symbolic computation techniques into a derived code which computes the function represented by the program and the derivative of this function. Also, **Odyssée** allows to compute adjoint codes, see [26], [12].

6.2 Some Comparison Tests

We have compared our approach to the standard one used in **Odyssée** in which the arrays are treated as atoms. We have made tests on three codes : **Bnrtl3** and **Srkphi** two programs coming from chemical engineering, **Trahad** a program that computes parameters of a three-dimensional schedule representing a differential equation of first order.

We have differentiated these programs by the three following methods: first by hand (BH) (except for **Trahad**), second (OA) by the standard method of **Odyssée**, and third (PA) by our approach. On those height codes, we have measured the memory requirements and the computing times by respectively the Unix command **size executable-file** and **time executable-file**.

Table 1. Time and Memory requirements on a SUN SS2

Programs	Number of lines	Number of dimensions	Size of Jacobian	Percentage sparsity	TIME			MEMORY		
					BH	OA	PA	BH	OA	PA
Bnrtl3	90	2	420	64	0.26	0.34	0.28	52584	57576	46144
Srkphi	122	2	1680	64	0.21	0.35	0.29	38544	85232	59176
Trahad	60	6	4E06	99	—	4.62	0.31	—	4.04E06	85600

Table 1 shows the promising aspect of this method. The Jacobians calculated for **Bnrtl3** and **Srkphi** are two-dimensional and consist of two parts: a lower triangular part and an identically zero part. The method (PA) stores and compute only the lower triangular part instead of entire matrices. In **Trahad**, the Jacobian is block form, where a block consists of between two and five diagonal vectors. Therefore, (PA) causes to store and compute from 2 to 5 three-dimensional arrays instead of storing and computing six-dimensional arrays with the method (OA). For these three programs, (PA) produces better results than (OA). The results obtained by the method (BH) show that the efficiency of this approach

is strongly related to the knowledge of the author on the original code. Generally, the results of (BH) are the best; however this approach is error-prone and writing by hand the derivative can take months.

7 Conclusion

In this paper, we have explored a method for computing fast derivatives by Automatic Differentiation using static program analysis. The method relies on the fact that dependence analysis between array regions enables us to provide informations about the structure of the Jacobian. Exploiting this information, we have proposed a way for generating more efficient code.

Static analyses cannot be exact in all cases and requires the use of conservative approximations. An extension of this work would be the definition of an appropriate tradeoff between static and dynamic approaches.

Acknowledgments. The authors would like to thank L. Hascoët for discussions on static program analysis and its applicability for optimizing Automatic Differentiation process.

References

1. A. Aho, R. Sethi and J. Ullman. *COMPILATEURS: Principes, techniques et outils*. InterEditions, 1989.
2. U. Banerjee. *Loop transformations for restructuring compilers: the foundations*. Boston, Dordrecht, London: Kluwer, 1993.
3. Christian H. Bischof, Ali Bouaricha, Peyvand M. Khademi, and Jorge J. Moré. *Computing gradients in large-scale optimization using Automatic Differentiation*. Preprint MCS-P488-0195, Mathematics and Computer Science Division, Argonne National Laboratory, June 1995.
4. C. H. Bischof, L. ROH, and A. J. MAUER-OATS. *ADIC: an extensible automatic differentiation tool for ANSI-C*. Software-Practice and Experience, vol. 27(12), 1427-1456, December 1997.
5. Aart J.C. Bik and Harry A.G. Wijshoff. *Automatic data structure selection and transformation for sparse matrix computations*. IEEE Transactions on Parallel and Distributed Systems, 7(2):109-126, 1996.
6. Vasanth Balasundaram and Ken Kennedy. *A Technique for summarizing Data Access and Its Use in Parallelism Enhancing Transformations*. Proceedings of the SIGPLAN conference on PLDI, Juin 1989.
7. Patrick Cousot and Radhia Cousot. *Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In Symposium on Principles of Programming Languages, pages 238-252, 1977.
8. Béatrice Creusillet and François Irigoin. *Exact vs. approximate array region analyses*. In Languages and Compilers for Parallel Computing, August 1996.
9. David Callahan and Ken Kennedy. *Analysis of Interprocedural side effects in a parallel programming environment*. Journal of Parallel and Distributed Computing, Juin 1988.

10. Thomas H. Coleman, and Jorge J. Moré. *Estimation of sparse jacobian matrices and graph coloring problems*. SIAM, Journal Numerical Analysis, Vol. 20, No. 1 February 1983.
11. Béatrice Creusillet *Analyses de régions de tableaux et applications*. Thèse de l'Ecole des Mines de Paris, 1996.
12. C. Faure, Y. Papegay. *Odyssée Version 1.6. The language reference manual*. Rapport Technique 211, INRIA, 1997.
13. A. Griewank, and G. Corlis. *Automatic Differentiation of algorithms: theory, implementation and application*. SIAM, 1991.
14. R. Giering. *Tangent Linear and adjoint model compiler, users manul*. unpublished information, Max-Planck Institut für Meteorologie Hambourg, Germany, 1996.
15. A. Griewank, D. Juedes and J. Utke. *ADOL-C, a package for the automatic differentiation of algorithms written in C/C++*. ACM Transactions on Mathematical Software, 22(2), pp: 131-167, 1996.
16. A. Griewank and S. Reese. *On the Calculation of Jacobian Matrices by the Markowitz Rule for Vertex Elimination*. Preprint MCS-P27-0491, Mathematics and Computer Science Division, Argonne National Laboratory, October 1991.
17. U. Geitner, J. Utke and A. Griewank. *Automatic Computation of Sparse Jacobians by Applying the Method of Newsam and Ramsdell*. to appear in: Computational Differentiation, Proceedings of the Second International Workshop, M. Berz et.al.eds, SIAM, Philadelphia, 1996.
18. Laurent Hascoët. *Specifications of Partita Analyses*. Version 1.4, INRIA, 1995.
19. Paul Havlack and Ken Kennedy. *An Implementation of Interprocedural Bounded regular Section analysis*. CRPC-TR-90063-S, March 1991.
20. Cristoph W. Kessler *On the Applicability of Program Comprehension Techniques to the Automatic Parallelization of sparse Matrix Computations*. Fachbereich IV - Informatik, Universität Trier, D-54286 Trier, Germany, 1997.
21. Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. *Transitive Closure of Infinite Graphs and its Applications*. University of Maryland Institute for Advanced Computer Studies Dept. of Computer Science, Univ. of Maryland, April 1994.
22. François Masdupuy. *Array indices relational semantic analysis using rational cosets and trapezoids* PhD thesis. Ecole polytechnique, France, 1993.
23. G. N. Newsam and J. D. Ramsdell. *Estimation of Sparse Jacobian Matrices*. SIAM J. Alg. Disc. Meth., vol.4(3), pp.404-417., 1983.
24. Sergio Pissanetsky. *Sparse Matrix Technology*. Academic Press, London 1984.
25. William Pugh and David Wonnacott. *An Exact Method for Analysis of Value-based Array Data Dependences*. Proc. of the Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing, June 1992.
26. Nicole Rostaing, Stéphane Dalmas, and André Galligo. *Automatic differentiation in odyssée e*. Tellus, 45A(5):558-568, 1993.
27. Robert Sedgewick. *Algorithms*. Adison-Wesley Publishing Company, 1984.
28. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley Publishing Company, 1991.

A New Solution to the Hidden Copy Problem^{*}

Deepak Goyal and Robert Paige

New York University, 251 Mercer Street, NY, NY 10012

{deepak, paige}@cs.nyu.edu

Fax # 212-995-4124, Tel # 212 998 3156

Abstract. We consider the well-known problem of avoiding unnecessary costly copying that arises in languages with copy/value semantics and large aggregate structures such as arrays, sets, or files. The origins of many recent studies focusing on avoiding copies of flat arrays in functional languages may be traced back to SETL copy optimization [Schwartz 75]. The problem is hard, and progress is slow, but a successful solution is crucial to achieving a pointer-free style of programming envisioned by [Hoare 75].

We give a new solution to copy optimization that uses dynamic reference counts and lazy copying to implement updates efficiently in an imperative language with arbitrarily nested finite sets and maps (which can easily model arrays, records and other aggregate datatypes). Big step operational semantics and abstract interpretations are used to prove the soundness of the analysis and the correctness of the transformation. An efficient algorithm to implement the analysis is presented. The approach is supported by realistic empirical evidence.

Our solution anticipates the introduction of arbitrarily nested polymorphic sets and maps into JAVA. It may also provide a new efficient strategy for implementing object cloning in Java and object assignment in C++. We illustrate how our methods might improve the recent approach of [Wand and Clinger 98] to avoid copies of flat arrays in a language of first-order recursion equations.

Keywords: Copy Optimization, Big Step Operational Semantics, Abstract Interpretation, Must Alias Analysis

1 Introduction

The problem of *hidden copies* arises in languages with *copy/value semantics* and large aggregate structures such as arrays, records, sets, or files. Consider, for example, the following two statements,

```
1      s := t;      -- assign set t to s
2      s with:= x;  -- add element x to s
```

^{*} This research was partially supported by Office of Naval Research Grant No. N00014-93-1-0924, and National Science Foundation grant CCR-9616993.

Here, variables s and t are set-valued and the operation $s \text{ with}:= x$ adds element x to set s . In a language with reference semantics, the element addition at Statement 2 above would cause a modification to the value of t also. However, such a side effect would be disallowed under copy/value semantics.

Copy/value semantics can be implemented by eager or lazy approaches. In an eager approach, we could implement Statement 1 by assigning a copy of set t to s ; Statement 2 could proceed by adding a copy of the value of x into set s in place. In a naive lazy approach Statement 1 could assign to s a pointer to the body of t , which makes s and t share the same location. Statement 2 could proceed by making a copy of s , and augmenting this copy in place with x . Making a copy of s avoids the side effect of updating other variables that share the location where s is stored.

Two interesting strategies have been considered to optimize the lazy approach. One strategy is to use static analysis to prove that an aggregate object is unshared by live variables at a program point, so that the object can be updated destructively at that point. Another is to maintain a dynamic reference count for each location that stores an aggregate object. An object at location L can be updated in place if the reference count at L is 1. We will describe a third strategy with the aim of facilitating more destructive (or in-place) updates than was possible before.

Copy optimization in functional languages with array updates is an important problem of intense current interest [11,10,7,29]. Little seems to be known, however, about Schwartz's extensive investigation of copy optimization for SETL in the 1970's [24]. Since we believe that this early work may be relevant to current research in the area, it is worth summarizing.

1.1 Motivation

SETL [22,25] is an imperative language with copy/value semantics for assignment and parameter passing, dynamic typing, and built-in finite sets, maps, and tuples of arbitrary depth of nesting. The hidden copy problem has been the major source of inefficiency in two generations of SETL compilers, which use a lazy copy strategy. If such a strategy is not implemented effectively, then hidden copies can potentially degrade program performance from $O(f(n))$ expected time to $O(f(n)^2)$ actual time. Such a slowdown has been actually observed even in small-scale SETL programs [3].

Although the hidden copy problem arises in different languages and language paradigms, it is also crucial to the more general goal, expressed by Hoare [8], of programming without pointers. Without a reasonable solution to this problem, Hoare's ideal cannot be achieved in any practical way. Currently without such a solution, we are forced to choose between two pragmatic, but unsatisfactory, compromises. One of these, exemplified by Ada and C++ is to retain copy/value semantics, but to rely on pointer-oriented programming to obtain efficiency. Another approach, taken by Java, is to assume reference semantics for aggregate objects.

1.2 An Unrealized Analytic Approach

Schwartz developed an interesting but complicated intra-procedural value flow analysis [23, 24] for SETL1 [25] in order to detect when destructive updates could be performed. His analysis determined an overestimate of the set of variables that at some level, no matter how deeply embedded, may share the same location. A destructive update to a variable v could be performed if no other variable that might share the location storing the value of v was live. His analysis aimed to be so fine-grained as to detect when destructive updates could be performed on components of aggregate structures.

Sharir [26] showed that value flow analysis did not fit any of the k -bounded monotone dataflow frameworks [28], and conjectured that extensions to interprocedural analysis would be too approximate to be useful. Based on these negative observations, Schwartz's approach was never implemented. Instead, SETL1 implemented a dynamic 'sticky' bit that was initially unset, but was set the first time that a location was shared and subsequently never unset. This solution was completely unsatisfactory, as was demonstrated by the performance of the Ada/Ed compiler [1].

Interestingly, researchers in the functional language community have shown that the kind of may-alias analysis for value flow combined with live variable analysis similar to Schwartz's approach is tractable even in the interprocedural case when datatypes are limited to flat arrays [10, 29]. However, it remains to be seen whether multi-level arrays and other aggregate data structures will confound this approach as they did for SETL.

1.3 A Dynamic Approach that didn't Work

In SETL2 [27] dynamic multi-level reference counts for each location storing an aggregate value (tuple, set, or map) are maintained. Highly restricted circularity of pointers ensures that when a location has a reference count greater than 1, then that location is *shared*, and cannot be updated unless it is first copied. Only when a location has reference count 1 can a destructive update be performed.

Since dynamic reference counts only degrade performance by a constant factor, this approach would be worthwhile if it can successfully prevent the loss of asymptotic factors. Unfortunately, the use of dynamic reference counts alone is no solution. The backend of Snyder's SETL2 compiler introduces so many compiler-generated temporary variables (which don't get garbage collected until the end of scope) that practically all data is shared at runtime.

For example, the standard SETL2 code generator implements indexed map assignment

```
f(a)(b) := d
```

using the following lower-level code,

```
t1 := f(a);    -- increment reference count for the location storing f(a)
t1(b) := d;    -- copy f(a) before update
f(a) := t1;    -- in place update if reference count(f) = 1
```

<pre> s := {} while P loop s with:= x end loop c with:= s </pre>	<pre> s := {} while P loop { s := copy(s) } s with:= x c with:= s end loop </pre>	<pre> s := t { s := copy(s) } while P loop s with:= x end loop c with:= s d with:= t </pre>
--	---	---

Fig. 1. Placement of copy operations

which makes a copy of map $\mathbf{t1}$ before it is updated in the second statement. However, if the location L that stores the value of \mathbf{t} is only shared by $\mathbf{f(a)}$, then the reassignment to $\mathbf{f(a)}$ makes such a hidden copy unnecessary. This problem can be solved by executing the command $\mathbf{f(a) := \omega}$ just before the indexed assignment to $\mathbf{t1}$. This omega assignment would decrement the reference count at L to 1, and allow a destructive indexed assignment to $\mathbf{t1}$.

1.4 Basis for a Solution

A solution to the hidden copy problem that assumes dynamic reference counts, may take into account the placement of copy operations and omega assignments (which can decrement dynamic reference counts). Since copy operations can be expensive, it is obvious that their placement needs attention. Figure 1 shows the most desirable place for copy operations to be performed. No hidden copies are needed for the first example. In the second example, a copy operation should be performed just before the element addition to \mathbf{s} within the while-loop. But in the third example, a single desirable hidden copy should be performed the first time only that the element addition is executed within the while-loop. Our solution is to abide by the SETL2 strategy of making a copy each time an update to an aggregate S cannot be performed destructively (based on a runtime check that the reference count for S is greater than 1). Interestingly, Schwartz's approach (with no dynamic reference counts) would handle the first two examples, but would fail on example 3.

The placement of omega assignments is more subtle. We say that a variable v is *live* at a program point p if the set of live uses of v is non-empty at p [42]. If a variable is not *live* at p , we say it is dead. In order for our transformation to be correct, we will insert omega assignments only to dead variables. Even though the cost of each omega assignment is relatively small, our tests show that the overuse of such assignments leads to suboptimal performance. Thus, our heuristic is to introduce omega assignments only when they are likely to facilitate destructive updates.

In Fig. 2, the first example shows that an omega assignment should be placed at a point where a dynamic reference count decrement will actually take place. The second example shows that an omega assignment should be placed at a point where it is not redundant. The third example generalizes the other two

<pre> s := t -- uses of s : -- s becomes dead ✓ while P loop ⊗ t with:= x end loop </pre>	<pre> while P loop s := t : -- uses of s -- s becomes dead ⊗ end loop ✓ t with:= x </pre>	<pre> while P loop if Q then : else s := t : -- uses of s -- s becomes dead ✓ endif ⊗ t with:= x end loop </pre>
--	---	--

Fig. 2. Placement of $s := \text{om}$ at \checkmark is more profitable than at \otimes

examples, and shows that an omega assignment should be made to a variable only if it shares a location with the variable being updated.

Our final solution uses a combination of static and dynamic techniques. The approach is to (1) augment the run-time structures with reference counts, and modify the run-time environment to perform in-place updates when the reference count is 1, and (2) use static analysis (must-alias and live-variable) to determine the appropriate places to insert omega assignments that reduce reference counts, *i.e.* for each update assignment s , look at the must-alias set $\text{alias}(s, v)$ for the variable v being updated; if all the other variables w in the must-alias set $\text{alias}(s, v)$ are dead, introduce statements of the form $w := \text{om}$ (thereby decrementing the reference count of location L that v is pointing to) just before assignment s . Even though we can't guarantee that setting all such variables w to omega will decrement the reference count of location L down to 1 (for example, an element of a live set-valued variable may be stored at location L), but at least we have a good chance. This is what we bet on.

1.5 Outline of the Paper

We report the following new results:

1. An effective copy optimization is given for the first time in a SETL-like language with dynamically typed finite sets and maps of arbitrary depth of nesting. These datatypes can conveniently model a wide range of aggregate datatypes including multilevel arrays and records.
2. In Section 2 we give big step operational semantics [12, 16] for (1) copy/value semantics, (2) a naive lazy copy strategy, and (3) an optimized strategy in which destructive updates are performed on data whose location is unshared. These three semantics are proved to be equivalent.
3. In Section 3 an abstract interpretation [5] is given to support the analysis at each program point of equivalence classes of *variables* that **MUST** share

the same location. Our *must-alias* analysis is made more accurate by the use of a congruence condition on the alias relations, which is similar to the right regularity condition of Deutsch’s *may-alias* analysis [6]. Our analysis is fine-grained enough to detect sharing of map components $\mathbf{f}(\mathbf{a})$. A safe approximation of this analysis is computed by an algorithm that runs in time $O(NV^2)$, where N is the number of program points, and V is the number of variables.

4. Finally, in Section 4 the must-alias analysis is combined with live variable analysis to form an effective intra-procedural transformation that selectively inserts safe omega assignments at appropriate program points according to a heuristic for profitability.

A simple but significant test of our approach involved the application of a local optimization to the SETL2 backend. Toto Paxia modified the SETL2 backend by adding an `o2` option that inserts omega assignments for dead variables for some simple cases such as the indexed assignment $f(a)(b) := d$. We tested the results of these optimizations on APTS [17], a 15000 line SETL2 program. APTS had originally been written in such a way as to solve a part of the hidden copy problem at source level. This involved the manual insertion of omega assignments to the source which not only increased the size of the program but also made it difficult to understand and maintain. However, once the new local optimizer was implemented, APTS was rewritten at a higher more normal level, and ran faster with the `o2` option than the old (hand optimized) version with or without the `o2` option. More significantly, APTS ran over 10 times faster with the `o2` option than without. More detailed experimental results are certainly desirable and a full implementation of our analysis along with comparative benchmarks are planned for the future.

2 Language

Variables : s, v, t, f, g, \dots

$\text{op} ::= \text{with} \mid \text{less}$

Expressions:

$e ::= \ni v$
 $\mid v$
 $\mid v_1(v_2)$
 $\mid \text{om}$
 $\mid \text{constants}$

Commands:

$\mathcal{L} ::= \text{read}(v)$
 $\mid v := e$
 $\mid v_1(v_2) := e$ (where v_1 must be distinct from v_2 and e)
 $\mid v \text{ op} := e$ (where v must be distinct from e)
 $\mid \mathcal{L}_1; \mathcal{L}_2$
 $\mid \text{if } e \text{ then } \mathcal{L}_1 \text{ else } \mathcal{L}_2 \text{ endif}$
 $\mid \text{while } e \text{ loop } \mathcal{L} \text{ endloop}$

Fig. 3. Definition of the language

Figure 3 defines the syntax of the kernel language \mathcal{L} to be transformed by copy optimization. Expression $\ni v$ returns an arbitrary element from set v . Expression

$v_1(v_2)$ represents single-valued map application. Commands v *with* $:= x$ and v *less* $:= x$ are for set element addition and deletion respectively. Without loss of generality we have omitted discussion of a much richer language that can be transformed into \mathcal{L} .

2.1 Copy/Value Semantics

Figure 4 gives an abstract formulation of the big-step operational copy/value semantics for \mathcal{L} . The domain of values is denoted by V_* , and the environment ρ maps variables to $(V_*)_{\perp}$ (the lifted domain of values), with \perp representing the undefined value. The elements of V_* represent canonical forms which are not described here but which can be obtained easily by lexicographic sorting or multiset discrimination (see [18]). External input constants and program constants are mapped to values in $(V_*)_{\perp}$ by an external procedure σ . Constants that belong to V_c include integers, sets $\{c_1, \dots, c_n\}$ of 0 or more constants, smaps $\langle c_1 \mapsto c'_1, \dots, c_n \mapsto c'_n \rangle$ of 0 or more pairs of constants, and the undefined atom **om**. Note that $\sigma(\text{om}) = \perp$. The initial environment is given by $\rho = \lambda x. \perp$. The semantics of **if-then-else**, **while** loop, and **statement sequence** are straightforward (see [30]), and are omitted.

In rules (II)–(III), judgment $\langle \rho, e \rangle \xrightarrow{e} u$ stands for the evaluation of \mathcal{L} expression e in environment ρ to obtain value $u \in (V_*)_{\perp}$. Judgment $\langle \rho, c \rangle \rightarrow \rho'$ represents evaluation of \mathcal{L} command c in environment ρ to obtain new environment ρ' .

In Fig. 4 we use the notation $\text{tag}(u)$ to represent an element of V_* , $\text{set}(u)$ and $\text{smap}(u)$ to represent set-valued and smap-valued elements of V_* , and u, u_1, u_2 to represent elements of $(V_*)_{\perp}$. Under this convention the reader should see that \perp can never be added to a set or smap. Note that the arbitrary selection expression $\ni S$ makes the semantics of \mathcal{L} nondeterministic.

2.2 Lazy Copy Semantics

Whereas the copy/value semantics defines what \mathcal{L} programs mean, we need an abstract formulation of the semantics of lazy copying in order to formally define our copy optimization in Section (3). Fig. 5 defines the semantics of \mathcal{L} with a naive lazy copying strategy. As before, we use simple domains to represent canonical forms, with details omitted. Domain V_{loc} is an infinite set of locations where data is stored. Environment ρ maps variables either to locations or integers, and *store* γ maps locations either to sets or smaps. In formal terms, $\text{domain}(\gamma)$ represents the subset of V_{loc} that has been allocated during execution, and $V_{loc} - \text{domain}(\gamma)$ represents unallocated locations. Types **inti**, **seti**, and **smapi** are the implementations of types **int**, **set**, and **smap**, respectively. The implementation of integers remains unchanged, but **seti** is implemented as a set of locations or integers. Similarly **smapi** is implemented as a set of pairs of locations or integers.

Domains

type τ	set V_τ
int	$\{int(x) : x = 0, 1, -1, 2, -2, \dots\}$
set	$\{set(s) : s \subseteq V_* \mid s < \infty\}$
smap	$\{smap(f) : f \subseteq V_* \times V_* \mid f < \infty \wedge f \text{ single-valued}\}$
*	$V_{int} \cup V_{set} \cup V_{smap}$

Rules

Environment $\rho : Vars \longrightarrow (V_*)_\perp$

V_c is the set of all syntactically correct constants

Externally defined procedure $\sigma : V_c \longrightarrow (V_*)_\perp$

$domain(u) \stackrel{def}{=} \{x : \exists y \mid [x, y] \in u \wedge y \neq \perp\}$

$$\langle \rho, c \rangle \xrightarrow{e} \sigma(c) \quad c, \text{ a constant} \quad (1)$$

$$\langle \rho, v \rangle \xrightarrow{e} \rho(v) \quad v, \text{ a variable} \quad (2)$$

$$\frac{\langle \rho, v \rangle \xrightarrow{e} set(u), \quad y \in u}{\langle \rho, \ni v \rangle \xrightarrow{e} y} \quad (3)$$

$$\frac{\langle \rho, v \rangle \xrightarrow{e} set(u), \quad u = \{\}}{\langle \rho, \ni v \rangle \xrightarrow{e} \perp} \quad (4)$$

$$\frac{\langle \rho, v_1 \rangle \xrightarrow{e} smap(u_1), \quad \langle \rho, v_2 \rangle \xrightarrow{e} tag(u_2), \quad [tag(u_2), u_3] \in u_1}{\langle \rho, v_1(v_2) \rangle \xrightarrow{e} u_3} \quad (5)$$

$$\frac{\langle \rho, v_1 \rangle \xrightarrow{e} smap(u_1), \quad \langle \rho, v_2 \rangle \xrightarrow{e} tag(u_2), \quad tag(u_2) \notin domain(u_1)}{\langle \rho, v_1(v_2) \rangle \xrightarrow{e} \perp} \quad (6)$$

$$\frac{\langle \rho, c \rangle \xrightarrow{e} u}{\langle \rho, read(v) \rangle \longrightarrow \rho[v \mapsto u]} \quad \text{where } c \text{ is an external constant} \quad (7)$$

$$\frac{\langle \rho, e \rangle \xrightarrow{e} u}{\langle \rho, s := e \rangle \longrightarrow \rho[s \mapsto u]} \quad (8)$$

$$\frac{\langle \rho, e \rangle \xrightarrow{e} tag(u), \quad \langle \rho, s \rangle \xrightarrow{e} set(w)}{\langle \rho, s \text{ op } := e \rangle \longrightarrow \rho[s \mapsto set(w \text{ op } tag(u))]} \quad (9)$$

$$\frac{\langle \rho, e \rangle \xrightarrow{e} \perp, \quad \langle \rho, f \rangle \xrightarrow{e} smap(g), \quad \langle \rho, a \rangle \xrightarrow{e} tag(x)}{\langle \rho, f(a) := e \rangle \longrightarrow \rho[f \mapsto smap(Q)]} \quad \text{where } Q = \{[y, z] \in g \mid y \neq tag(x)\} \quad (10)$$

$$\frac{\langle \rho, e \rangle \xrightarrow{e} tag(u), \quad \langle \rho, f \rangle \xrightarrow{e} smap(g), \quad \langle \rho, a \rangle \xrightarrow{e} tag(x)}{\langle \rho, f(a) := e \rangle \longrightarrow \rho[f \mapsto smap(Q \cup \{[tag(x), tag(u)]\})]} \quad \text{where } Q = \{[y, z] \in g \mid y \neq tag(x)\} \quad (11)$$

Fig. 4. Copy/value semantics of the Language

We use function *extract* to relate a *location* to the corresponding value that it represents in the copy/value semantics. Function *extract* is recursively defined as,

$$\begin{aligned} \text{extract}(\text{loc}(l), \gamma) &= \text{case } \gamma(\text{loc}(l)) \text{ of} \\ &\quad \text{seti}(s) \implies \text{set}(\{\text{extract}(x) : x \in s\}) \\ &\quad \text{smapi}(f) \implies \text{smap}(\{\text{extract}(x), \text{extract}(y)\} : [x, y] \in f\}) \\ &\quad \text{end case} \\ \text{extract}(\text{inti}(i), \gamma) &= \text{int}(i) \end{aligned}$$

Proposition 1. *Function “extract” is well defined for all locations in store γ at any point in the execution of an \mathcal{L} program*

The proof is a straightforward well-founded induction on the structure of locations in the store, and is omitted. The proof essentially asserts that no sequence of statements can create a set that is a member of itself, or create a smap that is an element of its own domain or range.

Rules (I2)–(I5) describe how program constants and external constants input by a read statement are evaluated. The method is to allocate an implementation of a canonical form $\sigma(c)$ of constant c within store γ . We use judgments of the form $\langle \gamma, \sigma(c) \rangle \rightsquigarrow (\gamma', l)$ to say that canonical form $\sigma(c)$ of constant c in the copy/value semantics is allocated into new store γ' at the new location l . Environment ρ maps integer constants to integers, and maps set-valued, or smap-valued constants to corresponding locations. We assume that all program constants are processed by calling a non- \mathcal{L} ‘system’ command *allocate*(c) for each constant just before program execution. Thus, the initial environment maps all program constants to corresponding locations or integers.

In Rules (I6)–(I26), judgment $\langle \rho, \gamma, e \rangle \xrightarrow{e} u$ stands for the evaluation of \mathcal{L} expression e in environment ρ and store γ to obtain either a location or an integer u . Note that preallocation of all program constants before program execution ensures that expression evaluation does not modify either environment ρ or store γ . Judgment $\langle \rho, \gamma, c \rangle \longrightarrow (\rho', \gamma')$ represents the evaluation of \mathcal{L} command c in environment ρ and store γ to obtain new environment ρ' and new store γ' . These rules essentially embody the idea that a simple assignment is implemented by just mapping the left-hand-side variable to the location corresponding to the right-hand-side, but that updates on sets and maps are implemented by modifying a copy. The rule for the read statement *read*(v) indicates that a new external constant c is input. If c is not an integer, it is allocated within the store at location l say, and the environment is modified so that v is mapped to l .

Although arbitrary selection operator $\ni S$ makes the semantics of \mathcal{L} non-deterministic, we can express the equivalence of copy/value and lazy copy semantics as follows.

Theorem 1. *For given program P , $\langle \lambda x. \perp, P \rangle \longrightarrow \rho_1$ iff $\langle \rho_{\text{init}}, \gamma_{\text{init}}, P \rangle \longrightarrow (\rho_2, \gamma)$ (where ρ_{init} and γ_{init} are obtained by preallocating the program constants in P) where for all variables v , $\rho_1(v) = \text{extract}(\rho_2(v), \gamma)$.*

The proof by rule induction (see for example [30]) is omitted here for the sake of brevity.

Implementation Domains

type τ	set V_τ
loc	infinite set of atoms $\text{loc}(l)$.
inti	V_{inti}
seti	$\{\text{seti}(s) : s \subseteq V_{\text{loc}} \cup V_{\text{inti}} \mid s < \infty\}$
smapi	$\{\text{smapi}(f) : f \subseteq (V_{\text{loc}} \cup V_{\text{inti}}) \times (V_{\text{loc}} \cup V_{\text{inti}}) \mid f < \infty \wedge f \text{ single-valued}\}$
*i	$V_{\text{inti}} \cup V_{\text{seti}} \cup V_{\text{smapi}}$

Rules

Assume V_c is the set of all the syntactically correct constants.

Environment $\rho : \text{Vars} \cup V_c \longrightarrow V_{\text{loc}} \cup V_{\text{inti}} \cup \{\perp\}$

Store $\gamma : V_{\text{loc}} \longrightarrow V_{\text{seti}} \cup V_{\text{smapi}}$

$\text{extract} : V_{\text{loc}} \cup V_{\text{inti}} \longrightarrow V_*$

$$\langle \gamma, \text{int}(x) \rangle \rightsquigarrow (\gamma, \text{inti}(x)) \quad (12)$$

$$\frac{l \in V_{\text{loc}} - \text{domain}(\gamma), \langle \gamma_i, u_{i+1} \rangle \rightsquigarrow (\gamma_{i+1}, x_{i+1}) \ \forall i = 0, \dots, n-1}{\langle \gamma_0, \text{set}(\{u_1, \dots, u_n\}) \rangle \rightsquigarrow (\gamma_n[l \mapsto \text{seti}(\{x_1, \dots, x_n\})], l)} \quad (13)$$

$$\frac{l \in V_{\text{loc}} - \text{domain}(\gamma), \langle \gamma_i, u_{i+1} \rangle \rightsquigarrow (\gamma'_i, x_{i+1}), \langle \gamma'_i, w_{i+1} \rangle \rightsquigarrow (\gamma_{i+1}, y_{i+1}) \ \forall i = 0, \dots, n-1}{\langle \gamma_0, \text{smapi}(\{[u_i, w_i] : i = 1, \dots, n\}) \rangle \rightsquigarrow (\gamma_n[l \mapsto \text{smapi}(\{[x_i, y_i] : i = 1, \dots, n\})], l)} \quad (14)$$

$$\frac{\langle \gamma, \sigma(c) \rangle \rightsquigarrow (\gamma', u)}{\langle \rho, \gamma, \text{allocate}(c) \rangle \xrightarrow{e} (\rho[c \mapsto u], \gamma')} \quad (15)$$

$$\langle \rho, \gamma, v \rangle \xrightarrow{e} \rho(v) \quad \text{for a variable } v \quad (16)$$

$$\langle \rho, \gamma, c \rangle \xrightarrow{e} \rho(c) \quad \text{for a constant } c \quad (17)$$

$$\frac{\langle \rho, \gamma, v_1 \rangle \xrightarrow{e} l_1, \ \gamma(l_1) = \text{smapi}(u_1), \ \langle \rho, \gamma, v_2 \rangle \xrightarrow{e} u_2, \ ([x, y] \in u_1 \wedge \text{extract}(x, \gamma) = \text{extract}(u_2, \gamma))}{\langle \rho, \gamma, v_1(v_2) \rangle \xrightarrow{e} y} \quad (18)$$

$$\frac{\langle \rho, \gamma, v_1 \rangle \xrightarrow{e} l_1, \ \gamma(l_1) = \text{smapi}(u_1), \ \langle \rho, \gamma, v_2 \rangle \xrightarrow{e} u_2, \ (\forall [x, y] \in u_1 \mid \text{extract}(x, \gamma) \neq \text{extract}(u_2, \gamma))}{\langle \rho, \gamma, v_1(v_2) \rangle \xrightarrow{e} \perp} \quad (19)$$

$$\frac{\langle \rho, \gamma, v \rangle \xrightarrow{e} l, \ \gamma(l) = \text{seti}(u), \ y \in u}{\langle \rho, \gamma, \exists v \rangle \xrightarrow{e} y} \quad (20)$$

$$\frac{\langle \rho, \gamma, v \rangle \xrightarrow{e} l, \ \gamma(l) = \text{seti}(u), \ u = \{\}}{\langle \rho, \gamma, \exists v \rangle \xrightarrow{e} \perp} \quad (21)$$

Fig. 5. Lazy Copy semantics (cont. on the next page)

$$\frac{\langle \rho, \gamma, \text{allocate}(c) \rangle \xrightarrow{e} (\rho', \gamma')}{\langle \rho, \gamma, \text{read}(v) \rangle \longrightarrow (\rho[v \mapsto \rho'(c)], \gamma')} \quad \text{where } c \text{ is an external constant} \quad (22)$$

$$\frac{\langle \rho, \gamma, e \rangle \xrightarrow{e} u}{\langle \rho, \gamma, s := e \rangle \longrightarrow (\rho[s \mapsto u], \gamma)} \quad (23)$$

$$\frac{\langle \rho, \gamma, e \rangle \xrightarrow{e} \text{tag}(u), \langle \rho, \gamma, s \rangle \xrightarrow{e} l, \gamma(l) = \text{seti}(w), \text{extract}(\text{seti}(w), \gamma) \neq \text{extract}(\text{seti}(w \text{ op } \text{tag}(u)), \gamma), l' \in V_{loc} - \text{domain}(\gamma)}{\langle \rho, \gamma, s \text{ op } := e \rangle \longrightarrow (\rho[s \mapsto l'], \gamma[l' \mapsto \text{seti}(w \text{ op } \text{tag}(u))])} \quad (24)$$

$$\frac{\langle \rho, \gamma, e \rangle \xrightarrow{e} \perp, \langle \rho, \gamma, f \rangle \xrightarrow{e} l, \langle \rho, \gamma, a \rangle \xrightarrow{e} \text{tag}(x), \gamma(l) = \text{smapi}(g), l' \in V_{loc} - \text{domain}(\gamma)}{\langle \rho, \gamma, f(a) := e \rangle \longrightarrow (\rho[f \mapsto l'], \gamma[l' \mapsto \text{smapi}(Q)])} \quad (25)$$

where $Q = \{[z, y] \in g | \text{extract}(z, \gamma) \neq \text{extract}(\text{tag}(x), \gamma)\}$.

$$\frac{\langle \rho, \gamma, e \rangle \xrightarrow{e} \text{tag}(u), \langle \rho, \gamma, f \rangle \xrightarrow{e} l, \langle \rho, \gamma, a \rangle \xrightarrow{e} \text{tag}(x), \gamma(l) = \text{smapi}(g), l' \in V_{loc} - \text{domain}(\gamma)}{\langle \rho, \gamma, f(a) := e \rangle \longrightarrow (\rho[f \mapsto l'], \gamma[l' \mapsto \text{smapi}(Q \cup [\text{tag}(x), \text{tag}(u)])])} \quad (26)$$

where $Q = \{[z, y] \in g | \text{extract}(z, \gamma) \neq \text{extract}(\text{tag}(x), \gamma)\}$.

Fig. 5. cont., Lazy Copy Semantics

2.3 Optimized Lazy Copy Semantics

The semantics of \mathcal{L} with an optimized lazy copying strategy is obtained from the unoptimized semantics by minor modification. This semantics sheds light on how our copy optimization can improve \mathcal{L} programs.

Define a relation $<\subseteq V_{loc} \times V_{loc}$ by the following inductive definition:

$$l_1 < l_2 \stackrel{\text{def}}{\iff} ((l_1 \in \text{range}(\rho) \vee (\exists l_0 | l_0 < l_1)) \wedge ((\gamma(l_1) = \text{seti}(s) \wedge l_2 \in s) \vee (\gamma(l_1) = \text{smapi}(f) \wedge \exists [x, y] \in f | (x = l_2 \vee y = l_2))))$$

In other words, $l_1 < l_2$ holds if l_1 is reachable from some variable by following a sequence of locations, and either l_1 corresponds to a set having l_2 as a member,

$$\frac{\text{Premises of Rule 24} \wedge \text{not shared}(l, \rho, \gamma)}{\langle \rho, \gamma, s \text{ op } := e \rangle \longrightarrow (\rho, \gamma[l \mapsto \text{seti}(w \text{ op } \text{tag}(u))])} \quad (27)$$

$$\frac{\text{Premises of Rule 25} \wedge \text{not shared}(l, \rho, \gamma)}{\langle \rho, \gamma, f(a) := e \rangle \longrightarrow (\rho, \gamma[l \mapsto \text{smapi}(Q)])} \quad (28)$$

where $Q = \{[z, y] \in g | \text{extract}(z, \gamma) \neq \text{extract}(\text{tag}(x), \gamma)\}$.

$$\frac{\text{Premises of Rule 26} \wedge \text{not shared}(l, \rho, \gamma)}{\langle \rho, \gamma, f(a) := e \rangle \longrightarrow (\rho, \gamma[l \mapsto \text{smapi}(Q \cup [\text{tag}(x), \text{tag}(u)])])} \quad (29)$$

where $Q = \{[z, y] \in g | \text{extract}(z, \gamma) \neq \text{extract}(\text{tag}(x), \gamma)\}$.

Fig. 6. New rules for optimized lazy copy semantics

or l_1 is a map having l_2 as a member of either its domain or range. For $l \in V_{loc}$, we also define

```

var_share( $l, \rho, \gamma$ ) =  $\{x \in \text{domain}(\rho) \mid \rho(x) = l\}$  -- variables sharing  $l$ 
set_share( $l, \rho, \gamma$ ) =  $\{l' : l' < l \wedge \gamma(l') = \text{seti}(-)\}$  -- sets sharing  $l$ 
domain_share( $l, \rho, \gamma$ ) =  $\{l' : l' < l \wedge \gamma(l') = \text{smap}(f) \wedge l \in \text{domain}(f)\}$ 
-- maps whose domain shares  $l$ 
range_share( $l, \rho, \gamma$ ) =  $\{[d, l'] : l' < l \wedge \gamma(l') = \text{smap}(f) \wedge f(d) = l'\}$ 
-- maps whose range shares  $l$  at specific domain points

```

Then the extent to which location l is shared in environment ρ and store γ is defined by reference count,

$$\text{refcount}(l, \rho, \gamma) \stackrel{\text{def}}{=} (|\text{var_share}(l, \rho, \gamma)| + |\text{set_share}(l, \rho, \gamma)| + |\text{domain_share}(l, \rho, \gamma)| + |\text{range_share}(l, \rho, \gamma)|).$$

Predicate $\text{shared}(l, \rho, \gamma) \stackrel{\text{def}}{\iff} \text{refcount}(l, \rho, \gamma) > 1$ decides whether l is shared, and is used in the optimized lazy copy semantics to facilitate destructive update of data stored at unshared locations.

The new semantics includes Rules (12)-(26) (from Fig. 5), adds new Rules (27)-(29) (see Fig. 6), and conjoins premise $\text{shared}(l, \rho, \gamma)$ to Rules (24)-(26).

Proposition 2. *Optimized lazy copy semantics preserves the semantics of unoptimized lazy copy semantics.*

In the next section we describe an alias analysis that computes a close approximation to sets

$$\text{var_share}(l, \rho, \gamma) \cup \{y : \exists l \in \text{range}(\rho) \mid y \in \text{range_share}(l, \rho, \gamma)\},$$

in order to justify copy optimization.

3 Must-Alias Analysis

This section describes the static data flow analysis used to compute a safe (sound) approximation to the must-alias relation $R \subseteq \text{Vars} \times \text{Vars}$ at each program point. A pair $\langle x, y \rangle$ belongs to R at a program point p , if we can guarantee that $\rho(x) = \rho(y)$ (wrt lazy copy semantics) for all possible environments ρ reaching program point p in any execution of the program. Note that the absence of a pair $\langle x, y \rangle$ from R does **not** imply that $\rho(x) \neq \rho(y)$ for all environments ρ reaching p . Note that $\langle x, y \rangle \notin R$ does not even imply the existence of at least one environment ρ reaching p such that $\rho(x) \neq \rho(y)$, since the relation R is only a safe approximation.

It is easy to see that any *must-alias* relation R is an equivalence relation. So R can be efficiently implemented as a *partition* of the set of variables Vars . P is a partition of Vars iff P is a set of nonempty mutually disjoint subsets of Vars whose union is all of Vars . The next subsection states some of the properties of partitions that will be useful in our analysis. A much more detailed description of these and many other properties can be found in [14].

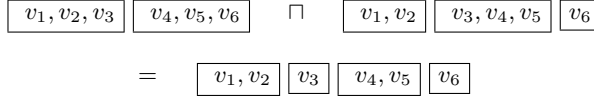


Fig. 7. Example of the \square (meet) operation for the Partition Lattice

3.1 Notation

Elements of a partition are sometimes called *blocks*. Let $\mathcal{P}(S)$ denote the set of all partitions over a finite set S . If $Q, P \in \mathcal{P}(S)$, then we say that Q is a *refinement* of P , denoted by $Q \sqsubseteq P$, iff $\forall b \in Q (\exists b' \in P | b \subseteq b')$. The partially ordered set $(\mathcal{P}(S), \sqsubseteq)$ has maximum element $\{S\}$ and minimum element $\{\{v\} : v \in S\}$, and, being finite, has the finite-descending-chain condition. The set of partitions form a Lattice, for which the meet (\square) is defined by

$$P \square Q \stackrel{def}{=} \{b_P \cap b_Q : b_P \in P, b_Q \in Q | b_P \cap b_Q \neq \{\}\}.$$

Figure 7 gives an example of the meet operation.

Each partition on a finite set S can be represented as a single-valued map from S to names of blocks, where each block in the partition is associated with a unique name. Let $name : blocks \rightarrow names$ map distinct blocks to distinct names. Then, the implementation $P' : S \rightarrow names$ of a partition P is given by,

$$P' = \cup_{b \in P} \{[x, name(b)] : x \in b\} \quad (30)$$

In the following sections, we will abuse our notation by associating the symbol b with both the block b (i.e. a subset of S) and $name(b)$. Thus, equation (30) could be rewritten as

$$P' = \cup_{b \in P} \{[x, b] : x \in b\}.$$

Given an implementation P of a partition of set S , $P(x)$ denotes the name of the block containing element x , and $[x]_P$ denotes the equivalence class of element x (i.e. the set of elements in the same block as x). Of course, since changing the name of any block does not change the underlying partition, we define a notion of equivalence between implementations of partitions by saying that two such implementations P and Q are equivalent if their underlying partitions are equal.

The following notation for overriding is very useful in describing partitions that are constructed from existing partitions by moving elements from one block, either to another existing block, or to a new block. Partition $P / \{[x, P(y)]\}$ denotes the partition obtained by moving element x to the block containing element y . Similarly, the partition obtained by moving x to a new block by itself is denoted by $P / \{[x, new(block)]\}$, where $new(block)$ returns a new block name. This notation for overriding can be extended to describe the movement of more than one variable into other existing or new blocks. Let $X : A \rightarrow names$ be single-valued, with $A \subseteq S$. Then, P/X represents a partition obtained from

P by moving elements in A into either existing or new blocks, and we have $P/X \equiv P'$, where $P'(y) = P(y)$ if $y \notin A$ and $P'(y) = X(y)$ otherwise. Finally, for $X_1 : A \rightarrow \text{names}$, and $X_2 : B \rightarrow \text{names}$ single-valued, with $A \cap B = \phi$, $X_1 \uplus X_2$ denotes the obvious union of these two maps. The disjointness of A and B ensures the single-valuedness of the resulting union.

3.2 Abstract Interpretation

This section describes an abstract interpretation approach based closely along the lines of [20], for computing a sound must-alias relation at each program point. As described in the previous subsection, the alias relation is a partition of the set of variables in the program. The set Vars comprises of all variables of the form v and $f(v)$ (smap application) that appear textually in the program. We use a nonstandard definition of Vars by including the variables of the form $f(v)$ appearing explicitly in the program. This enables us to maintain information such as whether $\langle f(v), s \rangle$ belongs to the must-alias relation or not. In fact, this is a key factor that enables us to do a reasonable must-alias analysis for arbitrarily nested sets and maps. Note, that even though only single-level map applications of the form $f(v)$ can appear textually in an \mathcal{L} program, multi-level map applications such as $f(x)(y)(z)$ or $f(g(h(x)))$ are easily translated into \mathcal{L} .

We consider an abstract interpretation framework where the environment-store pairs (from the lazy copy semantics) form the concrete domain and partitions form the abstract domain. We also define an abstraction function, that takes an element $\langle \rho, \gamma \rangle$ of the concrete domain to an element P of the abstract domain. The abstraction function merely states that two variables are mapped into the same block if and only if they are mapped to the same location in $\langle \rho, \gamma \rangle$.

Abstraction Function $\beta : \text{Env} \times \text{Store} \rightarrow \text{Partition}$

$$\begin{aligned} \beta(\langle \rho, \gamma \rangle) = P &\iff \forall v_1, v_2 \in \text{Vars} : \\ (\langle \rho, \gamma, v_1 \rangle \xrightarrow{e} l_1 \wedge \langle \rho, \gamma, v_2 \rangle \xrightarrow{e} l_2 \wedge (l_1 = l_2)) &\iff P(v_1) = P(v_2)) \end{aligned} \quad (31)$$

Just as statements act as $(\text{Env}, \text{Store})$ transformers in the concrete domain, similarly, statements act as partition transformers in the abstract domain. Figure 8 defines the abstract semantics for the statements in \mathcal{L} . We can read $\langle P, st \rangle \xrightarrow{a} P'$ as saying that the statement st transforms partition P to P' . The abstract semantics are most easily understood by looking at the examples in Fig. 9.

Suppose variables f and g are pointing to the same location. Then, for any variable u , $\rho(f(u))$ and $\rho(g(u))$ (if defined) will evaluate to the same location. In other words, if maps f and g are equal, then so are $f(u)$ and $g(u)$. Therefore, we make sure that the partitions in our analysis satisfy a congruence property that says that *if f and g are in the same block, then $f(u)$ and $g(u)$ should be in the same block*.

We now proceed with a step-by-step explanation of the rules for the abstract semantics described in Fig. 8.

$$\langle P, st \rangle \xrightarrow{a} P'$$

Notation: $m_1\{s\} = \{f \in \text{Vars} \mid f(s) \in \text{Vars}\}$
 $m_1[S] = \cup_{s \in S} m_1\{s\}$
 $m_2\{f\} = \{s \in \text{Vars} \mid f(s) \in \text{Vars}\}$
 $m_2[S] = \cup_{f \in S} m_2\{f\}$
 $m_f[S] = \{v \in \text{Vars} \mid f(v) \in S\}$

Case 1. If st is of the form $\text{read}(s)$, $s \text{ op} := e$, or $s := \exists v$:

$$\begin{aligned} P' = P / \{[s, \text{new}(\text{block})]\} \uplus \\ \uplus_{b \in P} (\text{let } n = \text{new}(\text{block}) \text{ in } \{[s(v), n] : v \in m_2\{s\} \cap b\}) \uplus \\ \uplus_{b \in P} (\text{let } n = \text{new}(\text{block}) \text{ in } \{[g(s), n] : g \in m_1\{s\} \cap b\}) \end{aligned}$$

Case 2. If st is of the form $s := t$:

$$\begin{aligned} P' = P / \{[s, P(t)]\} \uplus \\ \uplus_{b \in P \mid b \cap m_1[[t]_P] \neq \phi} \text{let } v = \exists \{v \in [t]_P \mid m_1\{v\} \cap b \neq \phi\} \text{ in} \\ \quad \text{let } f = \exists m_1\{v\} \cap b \text{ in } \{[g(s), P(f(v))] : g \in m_1\{s\} \cap b\} \uplus \\ \uplus_{b \in P \mid b \cap m_2[[t]_P] \neq \phi} \text{let } v = \exists \{v \in [t]_P \mid m_2\{v\} \cap b \neq \phi\} \text{ in} \\ \quad \text{let } f = \exists m_2\{v\} \cap b \text{ in } \{[s(g), P(v(f))] : g \in m_2\{s\} \cap b\} \uplus \\ \uplus_{b \in P \mid b \cap m_1[[t]_P] = \phi} (\text{let } n = \text{new}(\text{block}) \text{ in } \{[g(s), n] : g \in m_1\{s\} \cap b\}) \uplus \\ \uplus_{b \in P \mid b \cap m_2[[t]_P] = \phi} (\text{let } n = \text{new}(\text{block}) \text{ in } \{[s(g), n] : g \in m_2\{s\} \cap b\}) \end{aligned}$$

Case 3. The case for $s := f(t)$ is obtained from Case 2 by substituting t by $f(t)$.

Case 4. If st is of the form $f(t) := s$, then

$$\begin{aligned} P' = P / \{[f, \text{new}(\text{block})]\} \uplus \\ \{[f(v), P(s)] : v \in m_2\{f\} \cap [t]_P\} \uplus \\ \uplus_{b \in P \mid b \neq P(t) \wedge b \cap m_f[[s]_P] = \phi} (\text{let } n = \text{new}(\text{block}) \text{ in } \{[f(v), n] : v \in m_2\{f\} \cap b\}) \uplus \\ \uplus_{b \in P} (\text{let } n = \text{new}(\text{block}) \text{ in } \{[g(f), n] : g \in m_1\{f\} \cap b\}) \end{aligned}$$

Fig. 8. Abstract Semantics

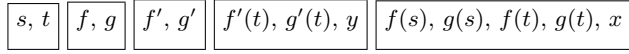
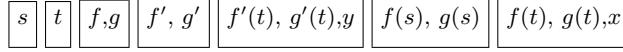
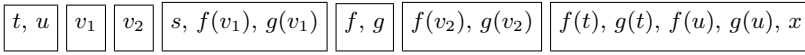
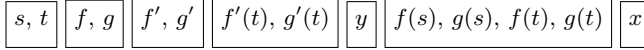
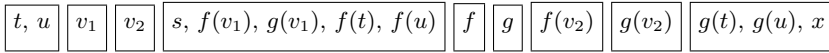
1. For statements of the form $\text{read}(s)$, $s \text{ op} := e$, $s := \exists v$:

The only variables whose values could be affected by these statements are of the form s , or $s(v)$, or $g(s)$. The effect of overriding P by $\{[s, \text{new}(\text{block})]\}$ is to move s to a new block (see example Fig. 9). Also, we see in the example that, if f and g were equal before the read, the fact that $\text{read}(s)$ cannot modify either f or g , guarantees that f and g (and, hence $f(s)$ and $g(s)$) must be equal after the read. This congruence effect is captured by

$$\uplus_{b \in P} (\text{let } n = \text{new}(\text{block}) \text{ in } \{[g(s), n] : g \in m_1\{s\} \cap b\}),$$

which says that for all variables in $\{g \in b \mid g(s) \in \text{Vars}\}$, we should move the corresponding variables of the form $g(s)$ into a block by themselves. A similar rule applied to variables of the form $s(v)$ and $s(u)$.

2. Statements of the form $s := t$:


 $\text{read}(s)$

 $t := s$

 $f(t) := s$

Fig. 9. Illustrative examples of the abstract semantics

The rule for this statement is more interesting. The effect of $\{[s, P(t)]\}$ is to move s into the block containing t . Suppose there exist variables v and $f(v)$ in the program such that $v = t$ before the assignment. Then after the assignment, we know that $s = t$. Therefore, if the variable $f(s)$ exists, it must equal $f(v)$ (by the congruence condition). The rule shown in Fig. 8 essentially captures this effect by considering two cases. In the first case, there exist variables v and $f(v)$ such that $(v \in [t]_P)$, so we move $f(s)$ to the block containing $f(v)$. In the second case, no such v exists, so $f(s)$ is moved to a new block. Again, we have to be careful in the second case to maintain congruence. The variables of the form $s(u)$ are handled in a similar manner.

3. Statements of the form $f(t) := s$:

Clearly this statement modifies the value of map f , which is moved to a new block. Looking at our example in Fig. 9, we see that since t and u are equal, we move both $f(t)$ and $f(u)$ to the block containing s . If v_2 is not in the same block as t , this does not mean that $s \neq t$. A modification to $f(t)$ could conceivably cause a modification to $f(v_2)$. This is why $f(v_2)$ moves into a block by itself. However, $f(v_1)$ is not moved to a block by itself, because it was already equal to s ; the statement $f(t) := s$ could not have possibly changed its value to something other than s . The rule given in Fig. 8 takes care of all these subtleties.

We now state a number of theorems dealing with the soundness of the abstract semantics. The proofs are long and can not be included here because of space constraints.

Theorem 2 (Local Soundness). *If st is a simple assignment statement, then*

$$\langle \rho, \gamma, st \rangle \longrightarrow \langle \rho', \gamma' \rangle \text{ and } \langle \beta(\langle \rho, \gamma \rangle), st \rangle \xrightarrow{a} P' \Rightarrow (P' \sqsubseteq \beta(\langle \rho', \gamma' \rangle))$$

Simply put, if the incoming partition is a safe approximation of the incoming environment-store pair, then the outgoing partition is also a safe approximation of the outgoing environment-store pair.

Theorem 3 (Monotonicity). *For any simple assignment statement st , and partitions P and Q such that $P \sqsubseteq Q$, we have*

$$\langle P, st \rangle \xrightarrow{a} P' \text{ and } \langle Q, st \rangle \xrightarrow{a} Q' \Rightarrow P' \sqsubseteq Q'$$

We define a path p from *start* (the beginning of the program) to a program point i as any sequence of simple assignment statements that lead from *start* to i . Note that, if there exists a cyclic path to i , then there are infinitely many possible paths to i (corresponding to 0 or more loops around the cycle). Let

$$paths(i) = \{\text{All paths to program point } i\}.$$

Furthermore, let us define f_p and f_p^* to be the concrete and abstract transfer functions for path p , i.e.,

$$f_p(\langle \rho, \gamma \rangle) = \langle \rho', \gamma' \rangle \stackrel{def}{\iff} \langle \rho, \gamma, p \rangle \longrightarrow \langle \rho', \gamma' \rangle, \text{ and}$$

$$f_p^*(Q) = Q' \stackrel{def}{\iff} \langle Q, p \rangle \xrightarrow{a} Q'$$

Then the *Meet-over-all-paths solution*, i.e., $\bigcap_{p \in paths(i)} \beta(f_p(\langle \rho_{init}, \gamma_{init} \rangle))$ is a safe approximation to the must-alias relation.

Theorem 4 (Global Soundness). *For all program points i in the program,*

$$\bigcap_{p \in paths(i)} f_p^*(\beta(\langle \rho_{init}, \gamma_{init} \rangle)) \sqsubseteq \bigcap_{p \in paths(i)} \beta(f_p(\langle \rho_{init}, \gamma_{init} \rangle))$$

In other words, the solution to the data flow analysis problem in the abstract domain is a safe approximation of the solution in the concrete domain.

The Proof follows from Local Soundness (Theorem 2) and Monotonicity (Theorem 3).

```

Init :  $\forall i = 0, \dots, n \ F(i) = \top$ 
Loop : repeat
       $\forall i = 1, \dots, n \ F(i) = \sqcap_{j \in \text{pred}(i)} f_{st(j,i)}^*(F(j))$ 
      until all  $F(i)$ 's stabilize.

```

Fig. 10. Kildall's Iterative algorithm for greatest fixed point computation

3.3 Algorithm

Let the program points be numbered from $0, \dots, n$, where program point 0 corresponds to the start of the program. We use $st(j, i)$ to denote the statement between program point j and its successor i . We use the results from [13] to claim that the meet-over-all-paths solution is safely approximated by the greatest fixed point solution of the following equation:

$$\begin{aligned}
 F(P_0, \dots, P_n) = & (g_0(P_0, \dots, P_n), \\
 & g_1(P_0, \dots, P_n), \\
 & \vdots \\
 & g_n(P_0, \dots, P_n))
 \end{aligned} \tag{32}$$

where $g_0(P_0, \dots, P_n) = \beta(\langle \rho_{init}, \gamma_{init} \rangle)$

and $g_i(P_0, \dots, P_n) = \sqcap_{j \in \text{pred}(i)} f_{st(j,i)}^*(P_j), \quad \forall i = 1, \dots, n.$

The above greatest fixed point can be computed by Kildall's iterative algorithm [15] (shown in Fig. 10).

Let V denote $|\text{Vars}|$ and let N denote the number of nodes in the control flow graph.

Proposition 3. 1. *Given partitions P_1 and P_2 , we can compute $P_1 \sqcap P_2$ in $O(V)$ time.*
 2. *Given a partition P , we can compute $f_{st}^*(P)$ for any simple assignment in $O(V)$ time.*

(1) can be proven directly from [14]. (2) has a direct implementation using simple data structuring and the techniques in [14].

Proposition 4. *Each iteration of Kildall's algorithm can be implemented in $O(N \times V)$ time.*

This follows directly from Proposition 3 assuming that the control flow graph has been processed so that each node has at most 2 predecessors and at most 2 successors [19].

Proposition 5. *Kildall's algorithm converges in at most $N \times V$ iterations. Thus the time complexity of the iterative algorithm is $O(N^2 \times V^2)$.*

```

worklist = {1, ..., n}
∀i = 1, ..., n  F(i) = ⊤
while (worklist ≠ ∅) loop
  remove an arbitrary element x from the worklist
  F(x) = ⋂y ∈ pred(x) fst(y,x)*(F(y))
  for z ∈ succ(x) loop
    if (z ∉ worklist) ∧ (⋂w ∈ pred(z) fst(w,z)*(F(w)) ⊆ F(x)) then
      worklist with := z
    endif
  end loop
end loop

```

Fig. 11. Algorithm based on Worklist strategy

The proof relies on the fact that the length of a strictly decreasing chain of partitions is bounded by V and that each iteration decreases the value of $F(i)$ at least at one program point i .

The time complexity of Kildall's algorithm can be improved by using a worklist strategy. The idea is to maintain a worklist of program points i that do not satisfy the condition $F(i) = \bigcap_{j \in \text{pred}(i)} f_{\text{st}(j,i)}^*(F(j))$. Figure 11 describes the modified algorithm.

Proposition 6. *The complexity of the worklist algorithm in Fig. 11 is $O(N \times V^2)$.*

The proof depends on showing that each program point can be added to the worklist at most V times. Furthermore, each time a program point is removed from the worklist, the processing takes $O(V)$ time. Hence the time complexity is $O(N \times V^2)$.

4 Copy Optimization Transformation

The problem of live variable analysis is well understood (see [21], for example). We have a live variable analysis that takes map variables such as $f(v)$ into account. This analysis is also proven correct using the framework of abstract interpretation. The abstract semantics and the proof of correctness are straightforward and are omitted.

The final solution is obtained as follows. For each update assignment, we look at the must-alias set for the variable being updated, and if all the other variables in this set are dead, these are assigned omegas just before the update. We are investigating improvements to this strategy in order to facilitate elimination of more copies.

5 Applications

Destructive array update optimization is critical for writing scientific codes in functional languages. Recently, Wand and Clinger [29] propose a solution for a call-by-value functional language based on interprocedural flow analysis for aliasing and liveness. Their optimization is based on the idea that if a *sound* live variable analysis can ensure that the array on which the update is being performed is not live after the update, then the update can be performed destructively.

Consider the definitions of a few functions in the simple first-order functional language considered by Wand and Clinger in Fig. 12. Their analysis tries to prove that the location corresponding to array B is always dead at the time when B is updated. However, they will not be successful in doing so for the simple reason that this is not true. This can be seen from the fact that the array A is live after the first call to function f from inside function g . Consequently, all updates on B will result in the creation of a new copy, although, if the calls to f are from function h , then the destructive updates on the arrays would still be legal. This problem is reminiscent of the problems that made Schwartz's value flow analysis [24] ineffective for solving the copy optimization problem in Setl.

```

fun Sum(A) = ...      //sum the elements of array A
fun f(B, i, j) = Sum(update(B, i, j))
fun g(A, i1, i2) = f(A, i1, 0) + f(A, i2, 0)
fun h(A, i1) = f(A, i1, 0)

```

Fig. 12. Small functional program fragment illustrating difficulties of a purely static analysis based approach to copy optimization

We believe that the key idea of using dynamic reference counts together with alias and liveness analysis could contribute to an effective solution to the problem. The use of reference counts, an interprocedural live variable analysis, and placement of dynamic reference count decrements could ensure that the reference count of array B is 1 when function f is called by function h , thereby allowing a destructive update.

Instead of using an approach based on dynamic reference counts, one could use a polyvariant static analysis to handle the example in Fig. 12. In this case we would use two versions of function f , one with an array copy and the other with a destructive update. The main drawback with this approach is a potential exponential blowup in the number of functions. For example, the call $k(X, X, 0, 1)$ of function $\text{fun } k(A, B, i_1, i_2) = h(A, i_1) + h(B, i_2)$, would also require both destructive and non-destructive versions of function h .

6 Conclusions and Future Work

This paper presents a new approach to copy optimization that trades potentially asymptotic costs of hidden copies for a constant factor overhead to maintain dynamic reference counts. The analysis and transformation are proved correct using formal semantics and abstract interpretations. A new low polynomial time algorithm is given to carry out the analysis.

It would be interesting to extend this work to the interprocedural case. We also want to explore partition refinement strategies found in [9] and [14] to improve the algorithm. It would be interesting to see how further optimization could cut down or eliminate dynamic reference counts. Our analysis deals with arbitrarily nested sets and maps that can be used to simulate a wide variety of datatypes such as records or even pointers (by the use of single-valued maps called *ref* and *deref*). We believe that our analysis and algorithmic techniques may apply to pointer analysis.

Acknowledgements. We would like to thank Toto Paxia for implementing a local optimizer for SETL2. We would also like to thank Mads Tofte and Zhe Yang for enlightening discussions about operational semantics. Finally, we would like to thank the anonymous referees for their valuable suggestions.

References

1. ADA UK News, Vol. 6, No. 1, pp. 14–15, Jan 1985.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1988.
3. J. Cai and R. Paige. Towards increased productivity of algorithm implementation. In *Proc. ACM SIGSOFT*, pages 71–78, Dec. 1993.
4. J. Cocke and J. Schwartz. *Programming Languages and Their Compilers*. Lecture Notes. Courant Institute, New York University, New York, 1969.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In *Proc. 4th ACM Symp. on Principles of Prog. Lang.*, pages 238–252, 1977.
6. A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *International Conference on Computer Languages*, pages 2–13. IEEE, 1992.
7. M. Draghicescu and S. Purushotham. A uniform treatment of order of evaluation and aggregate update. *Theoretical Computer Science*, 118:231–262, 1993.
8. C. A. R. Hoare. Data reliability. In *Proc. of the Intl. Conf. on Reliable Software*, pages 528–533, 1975.
9. J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971. Proc. Intl. Symp. on Theory of Machines and Computation.
10. P. Hudak. A semantic model of reference counting and its abstraction. In *Proc. 1986 ACM Symp. on Lisp and Func. Prog.*, pages 351–363. ACM, 1986.

11. P. Hudak and A. Bloss. Avoiding copying in functional and logic programming languages. In *Proc. 12th Annual ACM Symp. on Principles of Prog. Lang.*, pages 300–314. ACM, 1985.
12. G. Kahn. Natural semantics. In *Proc. STACS'87*. Springer-Verlag, 1987. Lecture Notes in Computer Science, Vol. 247.
13. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.
14. J. Keller and R. Paige. Program derivation with verified transformations – a case study. *CPAM*, 48(9-10), 1995.
15. G. A. Kildall. A unified approach to global program optimization. In *ACM Symp. on Principles of Prog. Lang.*, pages 194–206, 1973.
16. H. R. Nielson and F. Nielson. *Semantics with Applications, A formal introduction*. Wiley, 1992.
17. R. Paige. Viewing a program transformation system at work. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic*, volume 844 of *LNCS*, pages 5–24. Springer-Verlag, Berlin, Sep. 1994. Proc. Joint 6th Intl. Conf. on Prog. Lang. Impl. and Logic Prog. (PLILP) and 4th Intl. Conf. on Algebraic and Logic Prog. (ALP).
18. R. Paige and Z. Yang. High level reading and data structure compilation. In *Proc. 24th ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang.*, pages 456–469, Paris, France, 15–17 Jan. 1997.
19. B. K. Rosen, M. Wegman, and K. Zadeck. Global value numbers and redundant computations. In *ACM Symp. on Principles of Prog. Lang.*, pages 12–27, 1988.
20. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, January 1998.
21. D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *25th ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang.*, 1998.
22. J. Schwartz. *On Programming: An Interim Report on the SETL Project, Installments I and II*. New York University, New York, 1974.
23. J. Schwartz. Automatic data structure choice in a language of very high level. *CACM*, 18(12):722–728, Dec. 1975.
24. J. Schwartz. Optimization of very high level languages, parts I, II. *J. of Computer Languages*, 1(2-3):161–218, 1975.
25. J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.
26. M. Sharir. A few cautionary notes on the convergence of iterative data-flow analysis algorithms. Setl Newsletter 208, New York University, April 1978.
27. K. Snyder. The SETL2 programming language. Technical Report 490, Courant Insititute, New York University, 1990.
28. R. E. Tarjan. A unified approach to path problems. *JACM*, 28(3):577–593, July 1981.
29. M. Wand and W. D. Clinger. Set constraints for destructive array update optimization. In *Proc. IEEE Conf. on Computer Languages*. IEEE, May 1998.
30. G. Winskell. *The Formal Semantics of Programming Languages*. Foundations of Computing. MIT Press, 1994.

A Tutorial on Domain Theory in Abstract Interpretation

Roberto Giacobazzi

Dipartimento di Informatica, Università di Pisa

Corso Italia 40, 56125 Pisa, Italy

E-mail: giaco@di.unipi.it

URL: <http://www.di.unipi.it/~giaco>

Ph.: +39-50-887283 Fax: +39-50-887226

The success of abstract interpretation in the analysis of discrete dynamic systems stems from its simple, but nevertheless rigorously defined, underlying idea that the specification of the behavior of a system, e.g. a program, at different levels of abstraction, is an approximation of its formal semantics. This provides a variety of formal methods for specifying, modifying, and implementing program analysis tools. In particular, one of the most fundamental facts of abstract interpretation is that most interesting properties of the approximated semantics, like its precision, completeness, and compositionality, which may involve complex operators, fixpoints *etc.*, all depend upon the notion of *abstraction*, which is precisely and uniquely specified by the chosen abstract domain of properties. Because of the key role played by domains in abstract interpretation, any formal method to compare or transform abstract interpretations is therefore inherently based on corresponding methods to compare and transform abstract domains: *Modifying domains corresponds to modify abstract interpretations.*

In this tutorial we will see how the process of designing an abstract domain can be made systematic. The foundation of a theory of domains for abstract interpretation was fixed by Cousot and Cousot in their POPL'79 paper and, in more recent years, this has become a specific research field in abstract interpretation and static program analysis, as proved by the increasing interest in the community around domain-theoretic methods for domain design. We believe that these methods will play a key role in the next future development of abstract interpretation theory and applications.

The main idea behind the use of domain-theoretical methods in abstract interpretation is to specify abstract domains systematically from the specification of both the system we want to analyze and some basic properties of interest. We provide a guided tour in the wide literature concerning the theory and applications of systematic abstract domain design, by considering mainly those works that introduce general purpose methodologies for this task. All these considerations will allow us to sketch a kind of *domain theory* for abstract interpretation. This makes available a number of high-level facilities to tune program analysis in accuracy and costs, by refining, combining, compressing, simplifying, and decomposing domains in abstract interpretation. The result is an advanced algebraic framework, where semantics and analyses, at different levels of abstraction,

can be systematically designed, e.g., as solutions of recursive domain equations, and manipulated according to the need.

Although this theory of domains can be applied in all areas where abstract interpretation is applicable, from program analysis to comparative semantics and verification methods (e.g., type inference), program analysis still represents the typical target of applications for this theory. This is particularly important in, for instance, analysis of reactive systems and in abstract model checking, where optimal approximated systems can be systematically designed. We will give a quick overview on most recent applications of this theory of domains in static program analysis and comparative semantics, and we will address further research directions in this field.

Program Analysis *as* Model Checking of Abstract Interpretations

David Schmidt

Bernhard Steffen

Kansas State University * (USA) Universität Dortmund** (D)

Abstract. This paper presents a collection of techniques, a methodology, in which abstract interpretation, flow analysis, and model checking are employed in the representation, abstraction, and analysis of programs. The methodology shows the areas of intersection of the different techniques as well as the opportunities that exist when one technique is used in support of another. The methodology is presented as a three-step process: First, from a (small-step) operational semantics definition and a program, one constructs a *program model*, which is a state-transition system that encodes the program's executions. Second, abstraction upon the program model is performed, reducing the detail of information in the model's nodes and arcs. Finally, the program model is analyzed for properties of its states and paths.

1 Motivation

Recent research suggests that the connections between iterative data-flow analysis and model checking are intimate. The most striking application is the use of a model checker to calculate iterative bit-vector-based data-flow analyses [44, 47–49]; this application depends on the encoding of the bit-vector's bits as boolean propositions which are decided by model checking.

But the application of a model checker as the engine for flow analysis is wider than bit-vector problems on sequential programs. Recent work by Steffen and his colleagues has adapted the basic construction to an efficient treatment of parallel programs [29], and work by Dwyer and others [19–21] shows how problems formally solved with data-flow analysis techniques are more simply expressed and solved by model-checking techniques. And there are numerous examples of program validation done with flow analysis that in the present day would be termed model checking [5, 6, 33, 34, 39, 40].

* Department of Computing and Information Sciences, Manhattan, Kansas (USA), schmidt@cis.ksu.edu. Supported by NSF/DARPA CCR-9633388 and NASA NAG-2-1209.

** Lehrstuhl für Programmiersysteme, Universität Dortmund, GB IV, Baroper Str. 301, D-44221 Dortmund (Germany), steffen@cs.uni-dortmund.de

To understand the connections between flow analysis and model checking, a third component, abstraction, more precisely, abstract interpretation, must be used. Abstract interpretation provides the foundation upon which safe program representations rest, and understanding why model checking is a proper computational tool for flow analysis depends upon an understanding of the underlying abstraction techniques.

The purpose of this paper is to introduce a collection of techniques, perhaps a methodology, in which abstraction, flow analysis, and model checking are employed in the representation, abstraction, and analysis of programs. The methodology is meant to show the areas of intersection of the different techniques as well as the opportunities that exist when one technique is used in support of another.

The methodology is based on a three-step process: First, from a (small-step) operational semantics definition and a program, one constructs a *program model*, which is a state-transition system that encodes one (or many, or all) of the program's executions. Second, one might abstract upon the program model, reducing the detail of information in the model's nodes and arcs. Finally, one analyses the model for properties of its paths, e.g., live variables information, redundancy information, safety properties, etc. The methods used within the three stages include abstract interpretation, flow analysis, and model checking. In some places the tools are interchangeable; in others, one tool clearly plays a singular role.

The paper is structured as follows: reviews of iterative data flow analysis and model checking are undertaken first. Next, the abstract interpretation of operational semantics definitions is reviewed, and it is shown how to construct program models. Following this, abstraction on program models is presented. Correctness issues are reviewed, and finally, the extraction of program properties within the framework is examined. The paper concludes with a brief look at extensions to the basic technique.

2 Iterative Flow Analysis

Entities of Observation

Data-flow information can be considered an “entity of observation”; the set of such entities is typically structured as a complete lattice, where the ordering on the elements, \sqsubseteq , models the precision of information where ‘smaller’ means more precise, and the lattice operations \sqcup and \sqcap construct least upper bounds and greatest lower bounds of arbitrary subsets. In this paper we typically assume a lattice structure for sets of entities of observation.

Program Models

A traditional iterative data-flow analysis works from a program's control-flow graph, which is one instance of a *program model*:

A *program model* \mathcal{P} is a 5-tuple $(\mathcal{S}, \mathcal{A}, \rightarrow, s, E)$, where

1. \mathcal{S} is a set of *nodes* or *program states*.

Source program: `while even x do x:=x div2 od; y:=2`

Program model is: $(\mathcal{S}, \mathcal{A}, \rightarrow)$,
 where $\mathcal{S} = \{p_1, p_2, p_3, p_4\}$
 $\mathcal{A} = \{\text{even } x, \neg \text{even } x, x := x \text{ div } 2, y := 2\}$
 \rightarrow is defined below:

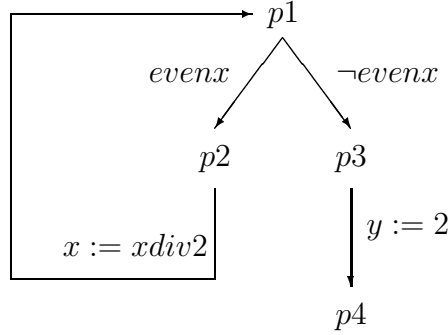


Fig. 1. An example program model for data-flow analysis.

2. \mathcal{A} is a set of *actions*, modelling elementary statements
3. $\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is a set of *labelled transitions*, i.e., modeling the flow of control.
4. s and E are the start state and the set of end states respectively.

We will write $p \xrightarrow{a} q$ instead of $(p, a, q) \in \rightarrow$, say that p is the *source* of the arrow and q is the *target*, and call p an *a-predecessor* of q and q an *a-successor* of p . The set of all *a*-predecessors and *a*-successors will be abbreviated by $Pred_a$ and $Succ_a$, respectively. Similarly, for subsets $A \subseteq \mathcal{A}$, we call p an *A-predecessor* of q and q an *A-successor* of p , if A contains an action a for which these properties hold, and we abbreviate the set of all *A*-predecessors and *A*-successors by $Pred_A$ and $Succ_A$, respectively.

Remark: A program model represents an execution of a program or a family of executions of a program. A classic example of a program model is a control-flow graph, which encodes all possible executions of a program; Figure 1 presents a control-flow graph represented as a program model.

A standard issue is that a program model might have an infinite state set, e.g., consider the program model that results from executing a program that loops while counting upwards by ones. For static analysis, we work with program models that have finite state sets, and we will assume that adequate abstraction techniques like widening [14] can be employed, when necessary, to ensure finiteness, and we will not explore such techniques any further.

Iterative Data Flow Analyses

Given a program model, one can define upon it an iterative data-flow analysis by defining a lattice of entities of observation, and for each transition in the program model, a transfer function. One uses the lattice and functions to define a set of equations, one per program point (or state) in the model. Solution of the equations yields the desired data-flow information. We classify data-flow analyses as being either forwards or backwards¹ and as being either \sqcup - or \sqcap -based.

In simple terms (see [25, 28] for a formal definition), one defines a backwards- \sqcup data-flow analysis of a program model by means of

- a set of entities of observation, D , partially ordered as a complete lattice.
- for each $A \in \mathcal{A}$, a *transfer function*, $f_A : D \rightarrow D$, that is monotonic on D : for $d, d' \in D$, $d \sqsubseteq d'$ implies $f_A(d) \sqsubseteq f_A(d')$.²
- for each program state, $p \in \mathcal{S}$, a *flow equation*,

$$val_p = \bigsqcup \{f_A(val_q) \mid (p, A, q) \in \rightarrow\}$$

The adequate *solution* to the set of flow equations can be determined by a least fixed-point computation.

Similarly, one can define a backwards- \sqcap -flow analysis by replacing the previous occurrence of \sqcup by \sqcap and computing the greatest-fixed-point solution of the equations. Finally, one can define a *forwards* analysis by swapping the occurrences of val_p and val_q in the flow equation scheme.

One example of a backwards- \sqcup data-flow analysis is live-variables analysis on a control-flow graph, which is formalized by

- $D = 2^{Var}$, where Var is the collection of the program's variables;
- $f_A(s) = UsedIn_A \cup (notModifiedIn_A \cap s)$, where $UsedIn_A$ defines those variables referenced in action A , and $notModifiedIn_A$ defines those variables that are not modified (assigned to) in A .

The flow equations follow from the above information. Figure 2 shows the live variables analysis for the program model in Figure 1.

3 Model Checking

Model checking is employed to validate properties of finite-state program models. This scope is sufficient for the purposes of this paper, but interested readers are referred to [4] for techniques covering infinite state models that explicitly model interprocedural structure. The considered properties are logical predicates

¹ We do not consider bi-directional algorithms here, which can, in fact, usually be decomposed into uni-directional components [30].

² It is common to demand that each f_A be *distributive*: for $D' \subseteq D$, $f_A(\bigsqcup D') = \bigsqcup \{f_A(d) \mid d \in D'\}$. Distributivity ensures that the fixed-point computation one performs on a set of flow equations calculates the same result as one obtains from a *meet-over-all-paths* analysis [27].

$$\begin{array}{ll}
 val_{p1} = f_{\{\text{evenx}\}}(val_{p2}) \cup f_{\{\neg\text{evenx}\}}(val_{p3}) & \text{where } f_{\{\text{evenx}\}}(s) = \{\mathbf{x}\} \cup (\{\mathbf{x}, \mathbf{y}\} \cap s) \\
 val_{p2} = f_{\{\mathbf{x}:=\text{xdiv2}\}}(val_{p1}) & f_{\{\neg\text{evenx}\}}(s) = \{\mathbf{x}\} \cup (\{\mathbf{x}, \mathbf{y}\} \cap s) \\
 val_{p3} = f_{\{\mathbf{y}:=2\}}(val_{p4}) & f_{\{\mathbf{x}:=\text{xdiv2}\}}(s) = \{\mathbf{x}\} \cup (\{\mathbf{y}\} \cap s) \\
 val_{p4} = \text{initialization-information} & f_{\{\mathbf{y}:=2\}}(s) = \{\mathbf{x}\} \cap s
 \end{array}$$

For *initialization-information* = $\{\mathbf{y}\}$, the analysis computes:

$$val_{p1} = \{\mathbf{x}\} \quad val_{p2} = \{\mathbf{x}\} \quad val_{p3} = \{\} \quad val_{p4} = \{\mathbf{y}\}$$

Fig. 2. Flow equations and solution for live-variables analysis

that typically discuss dependencies between occurrences of specific actions within paths in the model, e.g., “all paths starting from the current program point eventually include an *a* action” or “there exists a path including an *a* action infinitely often.”

The Syntax

As a logic for specifying static analysis algorithms, we will consider a variant of the temporal logic CTL [7] that includes a *parameterized* version of the “Henceforth” operator, the key for specifying *qualified safety* properties. This logic is particularly suited for expressing properties of states within a given (program) model.

The logic’s syntax is parameterized on a denumerable set \mathcal{B} of atomic propositions³ on states and a complete lattice of actions \mathcal{A} . Let β range over \mathcal{B} and $A \subseteq \mathcal{A}$:

$$\Phi ::= \beta \mid \Phi \wedge \Phi \mid \neg\Phi \mid [A]\Phi \mid \overline{[A]}\Phi \mid \mathbf{AG}_A \Phi \mid \overline{\mathbf{AG}}_A \Phi \mid$$

We write $p \models \Phi$ to denote that proposition Φ holds true at state p , and we say that p *satisfies* Φ .

The Semantics

Satisfaction is defined with respect to a given program model \mathcal{P} containing p according to the following intuition: $p \models \beta$ is assumed to be decidable, $p \models \Phi_1 \wedge \Phi_2$ if p satisfies both Φ_1 and Φ_2 . $p \models \neg\Phi$ if p does not satisfy Φ , and $p \models [A]\Phi$ if every one of p ’s A -successor states satisfies Φ . Note that this implies p satisfies $[A]\mathbb{f}$ exactly when p has no A -successors. Analogously, p satisfies $\overline{[A]}\Phi$ if every A -predecessor satisfies Φ . Thus in analogy, a program state p satisfies $\overline{[A]}\mathbb{f}$ exactly when p has no A -predecessors. Finally, $p \models \mathbf{AG}_A \Phi$ if Φ holds in every state reachable from p via A -transitions, and it satisfies $\overline{\mathbf{AG}}_A \Phi$, if exactly the same property holds for the inverted flow of control in the program model.

³ In this paper we simply consider three atomic propositions, *tt*, *start*, and *end*, which characterize the set of all states, the start state and the set of end states, respectively.

In general, any set of decidable characterizations of sets of states could be taken.

Remark: The difference between the standard Henceforth operator $\mathbf{AG} \Phi$ and the parameterized version $\mathbf{AG}_A \Phi$ is with respect to reachability: For $p \models \mathbf{AG} \Phi$ to hold, *all* states reachable from *all* transitions from p must satisfy Φ , whereas for $p \models \mathbf{AG}_A \Phi$, only those states reached from p by traversing transitions labelled by an action from A must satisfy Φ .

The formal semantic definition of the logic derived from the modal μ -calculus can be found in [47].

In the following we will write $[\cdot]$ or $\overline{[\cdot]}$ instead of $[\mathcal{A}]$ or $\overline{[\mathcal{A}]}$. Moreover, as usual, we can define the following duals to the operators of our language and the implication operator \Rightarrow by:

$$\begin{aligned} \text{ff} &= \neg tt & \mathbf{EF}_A \Phi &= \neg \mathbf{AG}_A \neg \Phi \\ \Phi_1 \vee \Phi_2 &= \neg(\neg \Phi_1 \wedge \neg \Phi_2) & \overline{\mathbf{EF}}_A \Phi &= \neg \overline{\mathbf{AG}}_A \neg \Phi \\ \langle A \rangle \Phi &= \neg [\overline{A}] (\neg \Phi) & \Phi \Rightarrow \Psi &= \neg \Phi \vee \Psi \\ \langle \overline{A} \rangle \Phi &= \neg [\overline{A}] (\neg \Phi) \end{aligned}$$

The Application

A crucial connection between iterative data-flow analysis and model checking was demonstrated by Steffen [47, 48], who noted the similarities between computing the results of a set of data-flow equations and computing the set of states that satisfied a modal mu-calculus specification. For example, the equation scheme that computes live variables,

$$Live_p = \bigcup \{ UsedIn_a \cup (notModifiedIn_a \cap s) \mid source(a) = p \}$$

can be transliterated into the following formula of our logic that expressess whether a variable, x , is live at a program point:

$$isLive_x = \mathbf{EF}_{\{a \mid a \neq mod_x\}} \langle use_x \rangle tt$$

This formulation is based on abstractions of the actions that annotate the arcs of a program model: Assignment statements, $v := e$, are abstracted to actions of the form, mod_v and use_u , for those variables, u , used in e ; tests, e , are abstracted to actions use_u , for those variables, u , used in e .

When one model checks the assertions, $p \models isLive_x$, for all variables x , one in effect computes the bit-vector solution for the flow equation $Live_p$. A standard iterative model checker would do this, e.g., for the variable x of Figure 1 by initially ‘marking’ all the states of the program model satisfying $\langle use_x \rangle tt$ with ‘true’, and, subsequently, stepwisely spreading this information to all states having a $\{a \mid a \neq mod_x\}$ -successor being marked ‘true’ until a fixed point is reached, i.e., no further state can be marked ‘true’ according to the described rules. For the considered example program the fixed point is reached already by the initialization procedure.

Technically, formulae of the logic considered here would be first translated into modal equational systems [10], a compact representation of the modal μ -calculus,

which have the appropriate granularity to directly steer the fixpoint computation process. $isLive_x$ would be translated into the following min-equation⁴

$$MIN: \quad isLive_x = \langle use_x \rangle tt \vee \langle \{a \mid a \neq mod_x\} \rangle isLive_x$$

The fixpoint computation needs one bit for each such equation. In particular, as expected, one bit (per program variable) is sufficient for checking liveness of variables.

Why Safety Properties

In this paper, we will explore the connection between flow analysis and model checking while focussing on the preservation of safety properties, i.e. properties which are guaranteed to hold for *all* reachable states. In other words these properties are characterized by holding *everywhere* along *all* program executions, and therefore correspond to the properties which can be specified using the parameterized Henceforth operator \mathbf{AG}_x .

$isLive_x$ is not a safety property, as it quantifies *existentially* over the execution paths⁵ as well as over the states of a given execution.⁶ However, this causes no real harm, as one is not interested in the fact whether a variable is live, but in the complement, as the detection of dead variables is a key for dead-code elimination, and this property is indeed a safety property. And indeed, as program transformations must be correct for all possible program executions, it is clear that they must be based upon information which is horizontally universally valid. In contrast, there seem to be relevant problems, like e.g. *down safety* [30], i.e., the property that a certain expression will be executed in every continuation of the program, which seem to ask for existential vertical quantification. Surprisingly, also these properties typically can be better formulated as a (parameterized) safety property than as a liveness property. This is due to the fact that the program executions we are interested in are typically *terminating* program executions. E.g., for down safety, we do not require the expression to be executed on paths ‘starving’ in a loop,⁷ but only on those which reach the end point of the program.⁸ This property can readily be expressed with a single parameterized Henceforth operator, if we assume that the end point of the program is characterized by the atomic proposition *end*:

$$\mathbf{AG}_{\{a \mid a \neq use_x\}}(\neg end \wedge \langle mod_x \rangle ff)$$

⁴ The distinction between min-equations and max-equations allows us to specify whether a minimal or a maximal fixpoint constitutes the desired solution.

⁵ This is apparent from the ‘*E*’ of the ‘*EF*’ operator, which means this ‘horizontal’ existential quantification.

⁶ This is apparent from the ‘*F*’ of the ‘*EF*’ operator, which means this ‘vertical’ existential.

⁷ Note that in control flow graphs, there is nothing to force a computation to leave a loop.

⁸ In some sense this can be regarded as a ‘partial correctness’ view to the problem.

$Store = Identifier \rightarrow Val$
 $Val = Nat$

$c ::= v := e \mid \mathbf{skip} \mid c_1; c_2 \mid \mathbf{if } e \mathbf{ then } c_1 \mathbf{ else } c_2 \mathbf{ fi} \mid \mathbf{while } e \mathbf{ do } c \mathbf{ od}$

$v := e, \sigma \xrightarrow{v:=e} [x \mapsto v]\sigma, \quad \text{where } [e]\sigma \Rightarrow v$

$$\frac{skip, \sigma \xrightarrow{skip} \sigma}{c_1; c_2, \sigma \xrightarrow{c} c_2, \sigma'} \quad \frac{c_1, \sigma \xrightarrow{c} \sigma'}{c_1; c_2, \sigma \xrightarrow{c} c_2, \sigma'} \quad \frac{c_1, \sigma \xrightarrow{c} c'_1, \sigma'}{c_1; c_2, \sigma \xrightarrow{c} c'_1; c_2, \sigma'}$$

The operational semantics of **while** is not given by a separate rule but deduced according to the following identity:

$\mathbf{while } e \mathbf{ do } c \mathbf{ od} \equiv \mathbf{if } e \mathbf{ then } c; \mathbf{while } e \mathbf{ do } c \mathbf{ od} \mathbf{ else skip fi}$

$$\frac{[e]\sigma \Rightarrow true}{\mathbf{if } e \mathbf{ then } c_1 \mathbf{ else } c_2 \mathbf{ fi}, \sigma \xrightarrow{e} c_1, \sigma} \quad \frac{[e]\sigma \Rightarrow false}{\mathbf{if } e \mathbf{ then } c_1 \mathbf{ else } c_2 \mathbf{ fi}, \sigma \xrightarrow{\neg e} c_2, \sigma}$$

Note: $[e]\sigma \Rightarrow v$ is defined by a relation, not given here.

Fig. 3. Small-step semantics for imperative language

Technically, we will present a *simulation-based* criterion for guaranteeing safety, which bridges the gap between the operational semantics of a program model on different levels of abstraction.

4 Operational Semantics and Abstractions

Our definition of program model includes not only control-flow graphs but also execution traces, their abstract interpretations, behaviour trees, and other kinds of abstract transition systems. Indeed, a program's control-flow graph can be simply regarded as a program model arising from an abstract interpretation of the program's semantics, where the program's store is abstracted to *nil*.

Small-Step Structural Operational Semantics

We assume that a programming language comes with a small-step structural operational semantics. An example of such a semantics appears in Figure 3. It is convenient to label each transition with the primitive command/expression that generates the transition.

Figure 4 displays part of the operational semantics of the program in Figure 1 generated from the small-step semantics rules. We call the derivation in the figure a *concrete computation*. Note that the concrete computation is also a program model.

Let $p1$ denote `while even x do x := x div2 od; y := 2`
 $p2$ denote `x := x div2; while even x do x := x div2 od; y := 2`
 $p3$ denote `y := 2`

Let $\sigma_{m,n}$ denote the store $[x \mapsto m][y \mapsto n]$

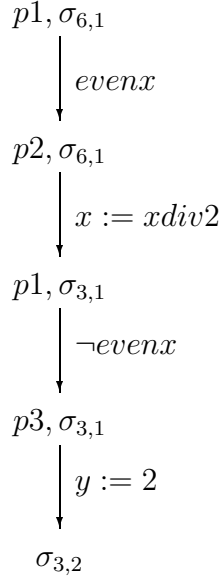


Fig. 4. Concrete computation

Standard Models

Given a programming language with a small step-semantics, the standard program model looks as follows:

- program states are the configurations appearing between transition steps (e.g., program point, store pairs, (p, σ))
- actions are the elementary statements and expressions of the language.
- transitions are defined by the small steps and are labeled with the corresponding primitive command/expression.
- the start state is the initial configuration, and the end states are those states having no successor.

As we will see, these standard models fully comprise the computations associated with a small-step semantics. In fact, a concrete computation and the corresponding standard program model are mutual simulations of each other in the formal sense established in Section 5. Thus they are semantically fully interchangeable for the purposes considered here, and are therefore a convenient abstract program representation.

Abstract Interpretation

For static analysis purposes, we wish to generate a finite program model that comprises all relevant concrete operational semantics executions. To do this, we employ the abstract interpretation methodology of Cousot and Cousot [14–16]: We replace the concrete domains, *Val* and *Store*, by abstract domains, *AbsVal* and *AbsStore*, and we compute a program’s operational semantics with the new domains to arrive at the corresponding program model. For ease of exposition, we require that the abstract domains be complete lattices. With the appropriate notion of abstraction of operations, the abstract domains will generate abstract program models that simulate the corresponding concrete computation models (cf. Section 5).

Galois Connections

To prove the simulation property, one must relate the abstract to concrete data domains by means of Galois connections⁹ [14]. If we think of an abstract data domain, *AbsVal*, as the “entities of observation,” then we must establish which subset of *Val* is denoted by an “entity” $a \in \text{AbsVal}$. We do so with a monotone mapping, a *concretization function*, $\gamma : \text{AbsVal} \rightarrow 2^{\text{Val}}$. Of course, there should be an inverse correspondence, a monotone $\alpha : 2^{\text{Val}} \rightarrow \text{AbsVal}$, the *abstraction function*; in particular, $\alpha(\{c\})$ identifies the element in *AbsVal* that “represents” $c \in \text{Val}$. We desire that (α, γ) form a Galois connection, because this implies $\alpha(\{c\}) = \sqcap \{a \mid c \in \gamma(a)\}$. It is well known that one adjoint of a Galois connection uniquely determines the other.

A useful intuition is that γ defines a binary “simulation” relation: for $c \in \text{Val}$, $a \in \text{AbsVal}$, $c \text{ safe } a$ iff $c \in \gamma(a)$. That is, c is simulated or safely approximated by a . Further, it is possible to begin with the binary relation, *safe*, and define a Galois connection from it: If *safe* is both *U-closed*, i.e., $c \text{ safe } a_1$ and $a_1 \sqsubseteq a_2$ imply $c \text{ safe } a_2$, and *G-closed*, i.e., $c' \text{ safe } \sqcap A$, where $A = \{a' \mid c' \text{ safe } a'\}$, then one obtains a Galois connection [43].

Because of these equivalences, we define a Galois connection by any one of a γ , α , or UG-closed relation in the sequel.

Control Flow Graphs

Here is the abstraction for control-flow graphs:

$$\begin{array}{ll} \text{AbsVal} = \{\text{nil}\} & \gamma : \text{AbsValue} \rightarrow 2^{\text{Val}} \\ \text{AbsStore} = \text{Identifier} \rightarrow \text{AbsVal} = \{\text{nil}\} & \gamma(\text{nil}) = \text{Val} \\ & \gamma : \text{AbsStore} \rightarrow 2^{\text{Store}} \\ & \gamma(\text{nil}) = \text{Store} \end{array}$$

⁹ Recall that a Galois connection is a pair of monotone functions, $(f: P \rightarrow Q, g: Q \rightarrow P)$, for complete lattices P and Q , such that $f \circ g \sqsubseteq \text{id}_Q$ and $\text{id}_P \sqsubseteq g \circ f$. The intuition is that $f(p)$ identifies p ’s most precise representative within Q (and similarly for $g(q)$).

Let $p1$ denote **while even x do $x := x \text{ div } 2$ od; $y := 2$**
 $p2$ denote $x := x \text{ div } 2$; **while even x do $x := x \text{ div } 2$ od; $y := 2$**
 $p3$ denote $y := 2$

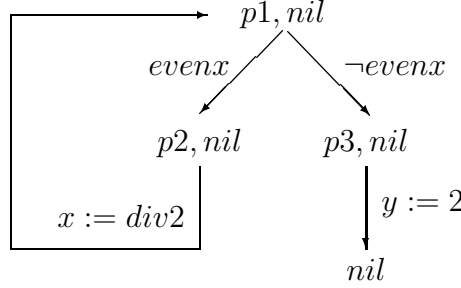


Fig. 5. Control-flow tree generated as an abstract computation

Figure 5 shows the standard program model generated from the control-flow abstraction of the program in Figure 4; it is of course an isomorphic representation of the usual control-flow graph. (Notice that the state, $(p1, nil)$, repeats in the model; hence, the arc from $(p2, nil)$ to $(p1, nil)$ can be written as a backwards arc.)

The next abstraction analyses a program's execution with respect to even-odd properties:

$AbsVal = \{\perp, e, o, \top\}$ (usual partial ordering)

$AbsStore = Identifier \rightarrow AbsVal$ (usual partial ordering)

$\gamma : AbsVal \rightarrow 2^{Val}$

$\gamma(\perp) = \{\}$

$\gamma(e) = \{n \in Value \mid n \text{ modulo } 2 = 0\}$

$\gamma(o) = \{n \in Value \mid n \text{ modulo } 2 = 1\}$

$\gamma(\top) = Val$

$\gamma : AbsStore \rightarrow 2^{Store}$

$\gamma(s) = \{s' \mid \text{for all } i, s'(i) \in \gamma(s(i))\}$

Abstract Operations

Once the concrete data domains are correctly abstracted by abstract data domains, one must abstract the operations that use the data domains and also the small-step semantics rules that use the operations. Assuming that an operation, $f_C : Val \rightarrow Val$ is a function, we say that f_C is *safely approximated* by function $f_A : AbsVal \rightarrow AbsVal$ iff

$$\text{for all } a \in AbsValue, \{f_C(m) \mid m \in \gamma(a)\} \subseteq \gamma(f_A(a))$$

Returning to binary simulation relations, the above definition of safe approximation is equivalent to this formulation: $c \text{ safe } a$ implies $f_C(c) \text{ safe } f_A(a)$.

In the case of even-odd analysis, the abstraction of the successor operation, $\text{succ}(n) = n + 1$, is naturally approximated by this function:

$$\begin{array}{ll} \text{succ}(o) \Rightarrow e & \text{succ}(\top) \Rightarrow \top \\ \text{succ}(e) \Rightarrow o & \text{succ}(\perp) \Rightarrow \perp \end{array}$$

Sometimes, imprecision can arise due to abstraction; consider the abstract version of division by 2, which produces an answer of \top when an even- or odd-valued number is divided by 2:

$$\text{div2}(\perp) \Rightarrow \perp \quad \text{div2}(a) \Rightarrow \top, \text{ for all } a \in \{e, o, \top\}$$

Abstract Operational Semantics

One uses the proved-safe abstract operations to define a small-step semantics for generating abstract program models. The intuition is that one instantiates the rule schemes to use the abstract operations rather than the concrete ones. Then, one uses the instantiated rules to generate a program model. But a standard problem is deciding the tests of conditional commands. (For example, how does the even-odd analysis decide the test of the conditional command in this example: $x := x \text{ div2}; \text{if even } x \text{ then } c_1 \text{ else } c_2 \text{ fi}$?) A solution is to rewrite the rules for the conditional as follows:

$$\frac{[e]\sigma \Rightarrow v, \text{ true} \sqsubseteq v}{\text{if } e \text{ then } c_1 \text{ else } c_2 \text{ fi}, \sigma \xrightarrow{e} c_1, \sigma} \quad \frac{[e]\sigma \Rightarrow v, \text{ false} \sqsubseteq v}{\text{if } e \text{ then } c_1 \text{ else } c_2 \text{ fi}, \sigma \xrightarrow{\neg e} c_2, \sigma}$$

This format assumes that the concrete *Boolean* domain is abstracted to a lattice, $\{\perp, \text{true}, \text{false}, \top\}$. If one wishes to retain the format of the original small-step rule schemes (and not perform abstraction on *Boolean*), then one must utilize relational definitions of abstract operations.¹⁰ But this topic will not be developed further in this paper.

When we use abstract semantics rule schemes like the two above, the program models contain nondeterministic branching (as would be the case in the example program, $x := x \text{ div2}; \text{if even } x \text{ then } c_1 \text{ else } c_2 \text{ fi}$), because the outcome of the test expression of a conditional command might not be uniquely *true* or *false*.

Figure 6 shows two abstract program models generated from the even-odd abstraction for the program in Figure 4. The first program model shows the result of analyzing a program where x starts as odd-valued; the second model results when x starts as even-valued. The examples show that more precise information about program paths can be elicited than what one obtains from a control-flow graph. Such models might be used if temporal properties of the program's paths must be validated, or if the inputs to a program are restricted to a particular range (e.g., input x is restricted to be odd-valued), or if better quality code can be

¹⁰ For example, a relational definition of div2 would be: $\text{div2}(v) \Rightarrow e$ and also $\text{div2}(v) \Rightarrow o$, for all $v \in \{o, e, \top\}$. Also, $\text{div2}(\perp) = \perp$. Similarly, a relational definition of the predicate, *even*, would be $\text{even}(o) \Rightarrow \text{false}$, $\text{even}(e) \Rightarrow \text{true}$, $\text{even}(\top) \Rightarrow \text{false}$, $\text{even}(\perp) \Rightarrow \text{true}$, $\text{even}(\perp) \Rightarrow \perp$.

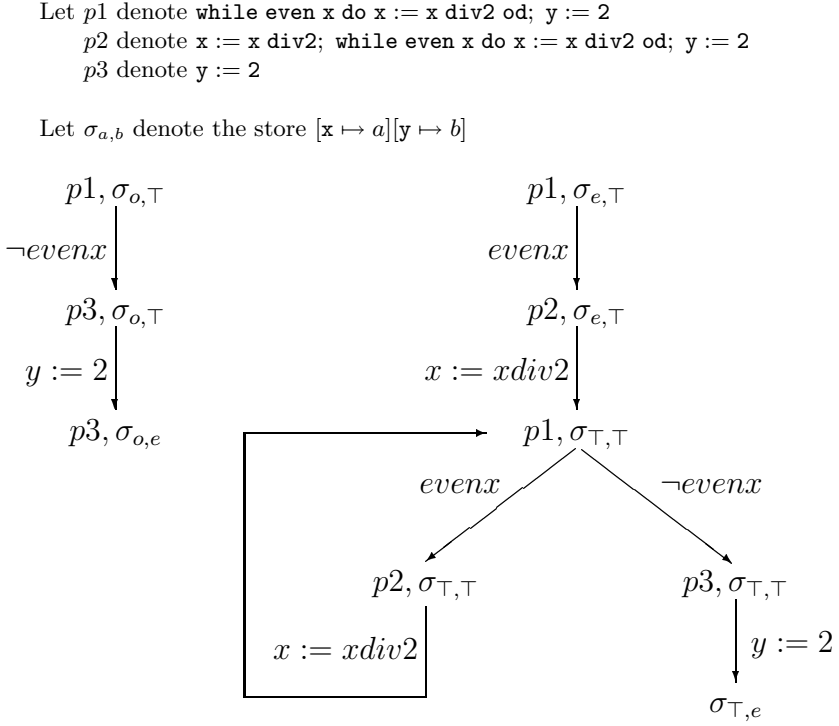


Fig. 6. Two abstract computations for even-odd analysis

generated when, say, the inputs to a command are even-valued. (If we analyze the example program where both x and y are initialized to \top , the resulting program model is isomorphic to the control-flow graph.)

Of course, we have the obligation of proving that the abstract program model is a safe approximation or simulation of the concrete model. Thus far, the safety properties enforced by Galois connections are “local” in that they relate concrete values to abstract values and concrete program states to abstract states. But there is still the burden of relating the *paths* in concrete program models (executions) to the paths in an abstract program model, that is, we must define precisely which set of concrete program models are approximated by an abstract program model. This must be established by a “global” safety property, to be defined in Section 5.

5 Simulations

One step of the “globalization” of the abstraction of a program model is the abstraction of the actions that label the arcs of the model. In the following, we will first introduce abstraction on actions by means of the prominent Use-Mod abstraction, and afterwards we will consider the correctness of such abstractions relative to the proofs of correctness of the abstractions on data domains and operations.

Abstraction of Actions

The development in the previous sections of abstraction of data domains and operations is standard. But we can abstract on source program syntax as well, more precisely, we can abstract on the actions that label the arrows of a program model

One example of abstraction of actions is the replacement of a command or expression by its Use-Mod information. For example, the Use-Mod abstraction of the command, $x := x + y$, would be $\{mod_x, use_x, use_y\}$, and the abstraction of the expression, **even** x , would be $\{use_x\}$.

Here is a formalization of Use-Mod abstraction; because Use-values are fundamentally covariant and Mod-values are fundamentally contravariant, we must take care in formulating the lattice, $UseMod$:

$$\begin{aligned} Mod &= \{isExpr\} \cup \{mod_x \mid x \in Identifier\} \\ Use &= \{use_x \mid x \in Identifier\} \\ UseMod &= \{\perp, \top\} \cup \sum_{i \in Mod} 2^{Use} \end{aligned}$$

The Use set is ordered by superset inclusion, and $UseMod$ is ordered as a disjoint union: For $(i_1, S_1), (i_2, S_2) \in UseMod$, $(i_1, S_1) \sqsubseteq (i_2, S_2)$ iff $i_1 = i_2$ and $S_2 \subseteq S_1$, and also $\perp \sqsubseteq v$ and $v \sqsubseteq \top$, for all $v \in UseMod$. (In the examples, we will continue to write (mod_x, S) as $\{mod_x\} \cup S$ and $(isExpr, S)$ as S .)

Next, we can define precisely how to map a single expression or assignment to its Use-Mod approximation:

$$\begin{aligned} \alpha_0 : Command \cup Expression &\rightarrow UseMod \\ \alpha_0(x := e) &= (mod_x, \{use_v \mid v \text{ appears in } e\}) \\ \alpha_0(e) &= (isExpr, \{use_v \mid v \text{ appears in } e\}) \end{aligned}$$

We define the lower adjoint, α , of a Galois connection in the expected way: $\alpha(S) = \bigsqcup \{\alpha_0(s) \mid s \in S\}$. The mate to α must be this γ :

$$\begin{aligned} \gamma : UseMod &\rightarrow 2^{Command \cup Expression} \\ \gamma(isExpr, S) &= \{e \mid v \in S \text{ implies } v \in e\} \\ \gamma(mod_x, S) &= \{x := e \mid v \in S \text{ implies } v \in e\} \\ \gamma(\perp) &= \{\}, \quad \gamma(\top) = Command \cup Expression \end{aligned}$$

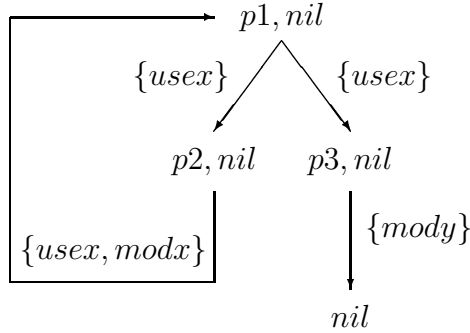


Fig. 7. Program model with actions abstracted to mod-use information

Model Construction and Abstraction of Actions

We intend to apply the *UseMod* domain to a live-variables analysis, and we do so in a surprising fashion: we first generate a program model where we do *not* abstract the set of actions, and we then replace actions, A , on the model's arcs by $\alpha(A)$. As an example, Figure 7 shows the replacement of the actions in the control-flow graph in Figure 5 by their abstractions: As we will justify in the next section, the revised program model is a safe simulation of all concrete executions of the program, hence it is possible to check safety properties on the model. In particular, one can validate that variable x is definitely dead at a program point, p , by performing the model check, $p \models \neg isLive_x$. This is the way that we formulate models for implementing bit-vector analyses.

Why did we take this approach of generating a program model first before abstracting its actions to Use-Mod information? After all, the standard abstraction methodology suggests that we apply the Use-Mod abstraction to the small-step semantics rule schemes first and generate a new set of small-step rule schemes that generate transitions based on Use-Mod “syntax.” But the Use-Mod abstraction discards so much syntactic structure that the resulting abstract rule schemes generate safe program models of worthless precision.

In such a situation, the way around the problem is to perform the above trick of abstracting the actions on an earlier existing, known-safe, program model. The result is a safe program model *for the set of concrete computations modelled safely by the earlier existing model*. In the above example, the Use-Mod abstraction gains precision by attaching itself to the control-flow-graph program model, which has a more precise branching structure than that obtained by using the small-step rule schemes based on Use-Mod syntax to generate a program model.

Having noted the above, we note there do exist examples in the literature where abstraction on syntax generates small-step rule schemes that *can* be used to generate useful program models. The best known example is the “Kleene-star abstraction” technique of Codish, Falaschi, and Marriott [12, 11], where the size

of an unfolded Prolog program is controlled by joining together goal clauses that use the same predicate symbol. A Prolog program is therefore abstracted into a syntax of regular expressions. Schmidt [42] uses a similar regular expression language to abstract the syntax of pi-calculus configurations. Approximations of program syntax by sets of context-free grammar rules have been done for functional programs by Giannotti and Latella [23] and for pi-calculus programs by Venet [52, 53].

Abstract Actions as Abstract Operations

In the above narrative, we did not place formal restrictions on the form of Galois connection that relates concrete actions to abstract actions. But in practice, the concrete actions represent commands—transfer functions—that update program state. When we write a transition, $c_1 \xrightarrow{A_c} c_2$, with a concrete semantics, and when we write the abstracted transition, $a_1 \xrightarrow{A_a} a_2$, in the abstract semantics, we assume that the “local simulation” described in the previous section is preserved by the abstraction of actions— $c_1 \text{ safe } a_1$ implies $c_2 \text{ safe } a_2$. This property is so desirable, we spend some time to study it.

Consider a concrete domain, D_c and the abstract domain, D_a , connected by a Galois connection (α, γ) . (Standard instantiations of these domains are 2^{Store} and $AbsStore$ of the previous section, but as the abstraction process may well be iterated, other instantiations are possible as well.)

The Galois connection induces a binary simulation relation, $\text{safe} \subseteq D_c \times D_a$ on the two domains (that is, for all $v \in D_c$, $u \in D_a$, $v \text{ safe } u$ iff $v \sqsubseteq \gamma(u)$, where $\gamma : D_a \rightarrow D_c$ is the upper adjoint of the Galois connection).

Next, assume that every action possesses a transfer function: For each action, $A_c \in \mathcal{A}_c$, let $\llbracket A_c \rrbracket_c : D_c \rightarrow D_c$ be its transfer function, and similarly, for each abstracted action, $A_a \in \mathcal{A}_a$, let $\llbracket A_a \rrbracket_a : D_a \rightarrow D_a$ be its transfer function. (Thus, for this story to be sensible, abstract actions must have transfer functions as well.)

The desired preservation property for action abstraction reads as follows:

$$\begin{aligned} &\text{for all } A_c \in \mathcal{A}_c, \text{ and for all } A_a \in \mathcal{A}_a, \text{ such that } A_a \text{ abstracts } A_c, \\ &\text{for all } v \in D_c, u \in D_a, v \text{ safe } u \text{ implies } \llbracket A_c \rrbracket_c(v) \text{ safe } \llbracket A_a \rrbracket_a(u) \end{aligned}$$

(By “ A_a abstracts A_c ,” we mean of course that $A_c \in \gamma_A(A_a)$ or equivalently, $\alpha_A(A_c) \sqsubseteq A_a$, using the Galois connection, (α_A, γ_C) , between the two action sets.)

If one decodes the consequent of the implication into the Galois connection between D_c and D_a , one obtains this inclusion at the concrete level D_c :

$$\bigsqcup \{ \llbracket A_c \rrbracket_c(v) \mid v \sqsubseteq \gamma(u) \} \sqsubseteq \gamma(\llbracket A_a \rrbracket_a(u))$$

which is equivalent to the following inclusion on the functional level, which we will use in the next section:

$$\alpha \circ \llbracket A_c \rrbracket_c \sqsubseteq \llbracket A_a \rrbracket_a \circ \alpha$$

From here on, we demand the following of the abstraction of concrete actions to abstract actions: *if A_a abstracts A_c , then $\alpha \circ \llbracket A_c \rrbracket_c \subseteq \llbracket A_a \rrbracket_a \circ \alpha$* . That is, the abstraction of actions preserves simulation on program states.

Remark Depending on the abstraction of program state, the preservation property just defined can be achieved trivially. Consider again the live-variables example in Figure 7, where both the “concrete” and “abstract” store sets are merely $\{\text{nil}\}$. (In the Figure, we do not care about the value of store, because it is the information on the arcs of the program model that matter.) The semantic transfer functions for both “concrete” and “abstract actions” are trivial.

We can make the live-variables example more interesting by letting concrete and abstract program stores tell us which expressions *will-be-used_e* for all expressions e . Assignments of the form $x := t$ operate on these entities backwards in the following fashion:

$$\llbracket x := t \rrbracket_c(S) = \{e \mid e \text{ is a subexpression of } t \text{ or } (e \in S \text{ and } [e]\sigma = [e]\sigma[[t]\sigma/x])\}$$

whereas the operation of the corresponding Use-Mod abstraction, which we assume here to be straightforwardly extended to expressions, is defined by

$$\llbracket \alpha_0(x := t) \rrbracket_a(S) = \{e \mid \text{use}_e \in \alpha_0(x := t) \text{ or } (e \in S \text{ and } \text{mod}_e \notin \alpha_0(x := t))\}$$

As long as we do not change the graph structure of the underlying program model, it is easy to verify that the Use-Mod abstraction computes a safe approximation of the concrete *will-be-used_e* information in the following sense: Whenever the abstract analysis tells us that an expression will be used in future then this holds also of the concrete analysis. (There are obviously situations where the reverse implication fails.)

The following section addresses the problem of changing graph structure, e.g. the collapsing of nodes, which is essential for fighting the state explosion problem.

Simulation Relations

The previous sections formalized in what sense a concrete value/operation/action is safely simulated or approximated by an abstract value/operation/action; these simulations are “local,” and it is time to extend simulation to program models, giving us a “global simulation” property that holds between program models, so that we can say precisely when a program model built with abstract operations and transitions safely describes a concrete program execution, which is represented by a concrete program model. Let, therefore

- $\mathcal{P}_c = (\mathcal{S}_c, \mathcal{A}_c, \rightarrow_c, s_c, E_c)$ and $\mathcal{P}_a = (\mathcal{S}_a, \mathcal{A}_a, \rightarrow_a, s_a, E_a)$ be two program models,
- D_c and D_a be complete lattices constituting the concrete respectively abstract domains
- $\llbracket \cdot \rrbracket_c : \mathcal{A}_c \rightarrow (D_c \rightarrow D_c)$ and $\llbracket \cdot \rrbracket_a : \mathcal{A}_a \rightarrow (D_a \rightarrow D_a)$ meaning functions, associating monotone/distributive transfer functions with their argument actions,

- (α, γ) be a pair of adjoints for D_c and D_a ,

Under these circumstances, we wish to formalize that \mathcal{P}_c is safely approximated by \mathcal{P}_a . The intuition is that every execution path in \mathcal{P}_c is imitated by one in \mathcal{P}_a . Since the arrows in the paths are labelled with actions, we must also verify that the corresponding labels on the corresponding arrows are compatible according to the abstraction function.

In this setting, the notion of safe approximation is essentially the property of simulation from concurrency theory. We formalize these intuitions precisely by means of a *simulation relation*, \mathcal{R}_α :

\mathcal{P}_a α -simulates \mathcal{P}_c , iff there exists a binary relation $\mathcal{R}_\alpha \in \mathcal{S}_c \times \mathcal{S}_a$ satisfying that

- $(s_c, s_a) \in \mathcal{R}_\alpha$
- for each pair $(s, t) \in \mathcal{R}_\alpha$, and all $A_c \in \mathcal{A}_c$ that:
 $s \xrightarrow{A_c}_c s'$ guarantees a transition $t \xrightarrow{A_a}_a t'$ with
 - $(s', t') \in \mathcal{R}_\alpha$ and
 - $\alpha \circ \llbracket A_c \rrbracket_c \sqsubseteq \llbracket A_a \rrbracket_a \circ \alpha$

It should be noted that the union of all simulation relations, \preceq_α , is again a simulation relation. Indeed, the largest possible simulation satisfying the above relation is exactly this union and is also the greatest-fixed point of the functional defined by the recursive definition [1, 16, 36, 35]. Thus a program model \mathcal{P}_a *globally simulates* a program model \mathcal{P}_c , iff $(s_c, s_a) \in \preceq_\alpha$, which we will in future write like $s_c \preceq_\alpha s_a$.

As promised in the previous section, we use the preservation of program state property, $\alpha \circ \llbracket A_c \rrbracket_c \sqsubseteq \llbracket A_a \rrbracket_a \circ \alpha$, to assert that concrete actions, A_c , are safely abstracted by abstract actions, A_a .

Our use of simulation to relate program models is a continuation of the machinery used in the earlier sections: Recall that a simulation relation, *safe*, can be used in safety proofs if it is G-closed, and *safe* defines a Galois connection if it is UG-closed. Although we do not prove it here, an α -simulation, \preceq_α , is U-closed, when one orders the set of abstract program models so that $s_{a1} \sqsubseteq s_{a2}$ when every path in s_{a1} exists in s_{a2} . If we refine the ordering to take into account the partial orderings on the information on the nodes and arcs, we can prove that \preceq_α is UG-closed. So, Galois connection notions carry forward to the top level of our levels of abstraction.

Guaranteeing Global Safety

As mentioned already, the primary notion of correctness (safety) is based on a transformational view: the information or property computed for a certain program point must be valid whenever a program execution reaches this point! Data flow analyses or model checking typically guarantee this criterion for a fixed level of abstraction. α -simulations provide a framework to guarantee this

correctness criterion for backwards safety properties in the context of abstraction in the following sense:¹¹

$s_c \preceq_\alpha s_a$ implies $s_a \models_a \phi \Rightarrow s_c \models_c \phi$ for all backwards safety properties ϕ .

Thus, under the considered circumstances, “global simulation” guarantees “local simulation”.

For forward analyses we have to consider backwards simulations¹² in order to obtain a similar result. Luckily, many abstractions on program models can be used for both forwards and backwards analyses. In these cases, all safety properties of the abstract model also hold of the concrete model.

Guaranteeing the Simulation Property

Two preservation properties are of interest, one concerning the operational semantics and one concerning program models.

Abstraction of the Operational Semantics. Once one has defined simulations/Galois connections for the concrete and abstract domains and also for the concrete and abstract actions, then one can instantiate the small-step semantics rule schemes with the concrete domains/actions and one can instantiate the rule schemes with the abstract domains/actions. As a corollary, one wants the result that the concrete semantics rules are “simulated” by the abstract semantics rules. And in fact, in [43], this result is indeed proved true for state-transition operational semantics. But for the richer format of Structural Operational Semantics, one requires a precise formalization of what it means to *instantiate* the rule schemes, a task going well beyond the scope of this paper. The reader is referred to [3, 16, 45] for the relevant machinery.

These result only address (forward simulation) and therefore only guarantee the preservation of backwards analysis for safety.

Abstraction of the Program Models. Besides abstraction on the domain and action level, which are admissible as long as corresponding Galois connections exist, the typical transformation steps are *collapsing*, i.e., identification of nodes on the abstract level while collecting the transition potential, and *unfolding*, i.e., copying of subgraphs with several entries. Whereas collapsing is clearly an abstraction, unfolding looks like a concretization at first sight, and, indeed, it is only an abstraction if it goes hand in hand with an abstraction on the domain level, where the concrete domain also distinguishes the different versions of duplicated nodes. We have:

¹¹ For simplicity we assume here that the formulas are identical on both levels of abstraction. In general this need not be true, e.g., it is often convenient to also abstract atomic propositions of the underlying logic.

¹² These are simply simulations of a program model with reversed transitions arcs—the “op” category, if you please.

- Locally correct abstraction on the domain and action side – characterized by the equation $\alpha \circ \llbracket A_c \rrbracket_c \subseteq \llbracket A_a \rrbracket_a \circ \alpha$ – together with any kind of collapsing guarantees the existence of a forwards *and* backwards simulation. Thus it supports the verification of *all* safety properties.
- Unfolding preserves forwards simulation only. In fact, it lies in the kernel of simulation, i.e. argument and result model simulate each other both ways. Thus unfolding can freely be used whenever only backwards analysis for safety is concerned.

These are only a few sufficient conditions for guaranteeing the preservation of simulation, and there is a huge potential for elaboration here and for enlarging the class of possible simulation-preserving constructions and transformations. Thus simulation provides a powerful background for establishing safety preserving abstractions.

6 Collecting Semantics

One can construct an abstract interpretation in order to extract from it a *collecting semantics*. A collecting semantics is information extracted from the nodes and paths of a computation graph. The classic, “first-order” [38] collecting semantics extracts the states from a computation graph: For a program model (or in general, a graph), g , its first-order collecting semantics has form $coll_g : ProgramPoint \rightarrow 2^{V^{val}}$ and is defined

$$coll_g(p) = \{v \mid (p, v) \text{ is a node in } g\}$$

Constant-propagation and type-inference analyses calculate answers that are first-order collecting semantics.

Another form of collecting semantics is “second order” or path based; it extracts paths from the model. The set of paths that go into a program point, p , is defined

$$fcoll_g(p) = \{r \mid r \text{ is a path in } g \text{ from } root(g) \text{ to some } (p, v)\}$$

and the set of paths that emanate from p is

$$bcoll_g(p) = \{r \mid r \text{ is a maximal path in } g \text{ such that } root(r) = (p, v)\}$$

The two collecting semantics are named *fcoll* and *bcoll* because they underlie the information one obtains from forwards and backwards iterative flow analyses, respectively. For example, a live-variables analysis calculates (properties of) $bcoll_g$, and an available expressions analysis calculates (properties of) $fcoll_g$.

The above forms of collecting semantics are “primitive” in the sense that no judgement about the extracted information is made. In practice, the information one desires from a data-flow analysis is a judgement whether some property holds true of the input values to a program point or of the paths flowing into/out of a program point. (For example, a live-variables analysis makes judgements about which variables are live for program point p in paths in $bcoll_g(p)$.)

To include such judgements, Cousot and Cousot [17] suggest that a model’s collecting semantics can be a set of properties expressed in some logic, \mathcal{L} . Given a program model, g , and a proposition, $\phi \in \mathcal{L}$, we write $\phi \in \llbracket g \rrbracket$ if ϕ holds true of g .

For judgements to be useful, we require a *weak consistency* property of \preceq_α and \mathcal{L} :

$$g_C \preceq_\alpha g_A \Rightarrow (\text{for all } \phi \in \mathcal{L}, \phi \in \llbracket g_A \rrbracket \Rightarrow \phi \in \llbracket g_C \rrbracket)$$

That is, any property possessed by an abstract model, g_A , must also hold for a corresponding concrete model, g_C . By tightening the two implications in the above formula into logical equivalences, we obtain weak completeness and strong completeness, respectively. The former is studied in [17]; the latter is employed to justify correctness of reductions of state spaces in concurrency theory [9, 18, 32].

With this viewpoint, the extraction of collecting semantics—flow information—is in fact the extraction of properties from a program model. This suggests that there is little difference between the collection of data-flow information and the validation of logical safety and liveness properties from a program model.

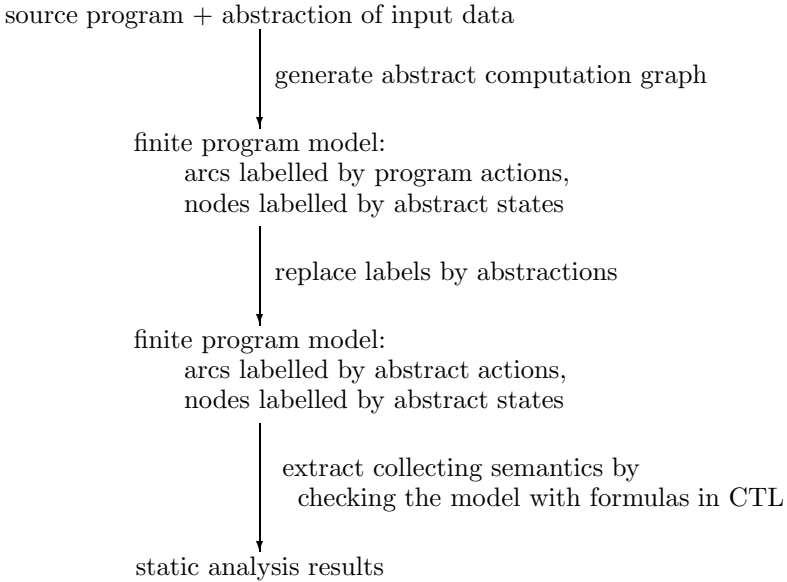
Since validation of properties of finite-state models can be done with a model checker, model checking becomes the natural tool for computing collecting semantics,¹³ where collecting semantics is encoded as logical properties, as suggested above. In the next section, we employ a model checker to calculate collecting semantics encoded as properties in CTL.

7 An Analysis Methodology

The previous sections have introduced the necessary techniques for construction, abstraction, and analysis of program models. When the techniques are organized into a methodology for program interpretation and static analysis, the following three stage approach to abstraction appears, a framework that we have used in practice:

¹³ In fact, iterative model checkers are directly computing a collecting semantics in this sense.

Three Stages of Abstraction



The first stage in the picture might be termed “abstraction of state,” because it is typical that a program’s input data and store are abstracted to “entities of observation” when generating the finite program model. The abstraction might be a trivial one, where both input data and program store are abstracted to *nil*, or it might be nontrivial, where input data are abstracted to *even-or-odd* and the store is a vector of variables with *even-odd* values.

The second stage in the above picture might be named “abstraction of action,” because the actions (primitive statements) in the program are abstracted to a property of interest. The abstraction might be trivial, where actions are left as they are, or it might be nontrivial, where expressions and commands are abstracted to Use-Mod information, like that used for bit-vector analyses. The result of the second stage is a finite program model whose paths show the order in which properties of actions might occur.

With this perspective, we note that the purpose of the first two stages in the picture is a complete removal of a program’s “syntactic overhead,” as contained in the program’s source syntax and its operational semantics rules. The result, the abstracted program model, contains the data- and control-essence of the program, which can be queried by a model checker to elicit the program’s semantic properties.

If the third stage in the picture must be given a name, it would be called the abstraction of the program model itself, because one queries the program model

about the information at its states (nodes) and its actions (the labels on the arcs of its paths), and one collects this information into a static analysis, which is rightly an abstraction of the program model itself.

Querying and Collecting Data Flow Information

As suggested by the previous section and Section 3, the queries one makes of the program model can be given in a temporal logic such as CTL, or better in the version of CTL with the parameterized Henceforth operator considered here.¹⁴ For example, one can assemble a complete summary of dead-variable information¹⁵ by asking the following question, for each state (node) of a program model whose arcs are labelled with Use-Mod information

$$isDead_x = \mathbf{AG}_{\{a \mid a \neq use_x\}}(\neg end \wedge \langle mod_x \rangle ff)$$

for each variable, x , in the source program. The answers to all these queries are collected together; they constitute a dead-variables analysis.

CTL is a good language for stating queries about a program model, because it is a language for expressing patterns of information that one encounters when one traverses the paths of a model. Indeed, most static analyses are defined to collect information about the patterns of states and actions one encounters along a program's execution paths.

For example, dead-variables analysis is a static analysis that calculates, for each program point, the paths that connect a program point, p , to uses of variables. (If a variable, x , is not used at the end of any such path from p , it is “dead” at p .) When a bit-vector-based dead-variables analysis does a similar calculation, it attaches a bit vector, one bit per program variable, to each program point, p . The bit for variable x at p is set “off” if there exists a path from p to a use of x , which makes the variable live. Thus, the bits in a bit-vector encode a set of yes-no answers to queries about the state of variables along program paths, and a variable is dead at p if its bit maintains its initial value true.

A similar story can be told for other bit-vector-based, iterative data-flow analyses—the bit vectors encode the answers to queries about the paths leading out of (or the paths leading into) the program points in a program model.

In this way, bit-vector analyses are neatly described by our framework. But the framework can do more than handle just bit-vector-based data-flow analyses.

Beyond Bitvector Analyses

1. At the first abstraction stage, abstraction of state, one can use nontrivial abstractions—even-odd properties, sign properties, finite-ranges-of-values properties, constant-value properties—to generate program models more precise than just a control-flow graph. The “precision” in this case might mean that the program model is smaller than the control-flow graph, which may

¹⁴ We will simply write CTL also for this parameterized version in the rest of this paper.

¹⁵ This information is dual to the live-variable information.

be the case when the additional information suffices to evaluate conditionals, or that the program points are “split” (duplicated with different store values at different nodes), producing what the partial-evaluation community calls a *polyvariant analysis* [26]. The same idea also underlies the property-oriented expansion proposed in [49].

In addition, a program’s input data can be restricted to reflect a precondition. For example, the first program model in Figure 6 shows the precision one gains when one knows from the outset that input variable x must be restricted to an odd number.

2. At the second abstraction stage, abstraction of actions, one can do abstraction of actions to other forms of information besides Use-Mod information. A standard example of action abstraction is seen in the behaviour trees generated by CCS-expressions [35], where actions are abstracted to “channels.” Also, one might abstract all actions to *nil*; which would be appropriate for calculations of “first-order collecting semantics,” as suggested in Section 6.
3. At the third stage, the extraction of collecting semantics, one might use one of a variety of logics to ask questions of the program model. Alternatives to CTL, such as CTL*, LTL, mu-calculus, and Büchi automata, can also be used this way, and the next section gives examples.

Finally, one can use the program logic to validate safety properties about paths in the program model. That is, the CTL formula we use to extract collecting semantics is also a logical proposition stating a “safety property” of a path. And of course, program validation of real safety properties can be performed in the very same framework that we have promoted for data-flow analysis. Of course, the dual is well known: data-flow analysis has been used for program validation [19, 33, 34, 39, 40], but the point has not been made as strongly as here, where we use the *very same logic* for both flow analysis and program validation.

Another benefit from the framework presented here is the clarification it gives to the stages of a static analysis—one can, at least conceptually, build a program model *first* and extract static information from it *second*.¹⁶ Often, these two stages are intertwined in presentations of static analyses, making correctness proofs harder to write and extensions harder to implement.

8 Discussion

We now discuss several applications where our methodology led to improved solutions of static analysis problems.

State Explosion

Consider property validation for large programs or for systems of communicating processes: If one generates a program model for a system of processes, the model’s

¹⁶ This does of course not exclude an ‘on-the-fly’ implementation, where the model is only constructed on demand.

state space quickly becomes intractable. One solution is to construct a naive program graph model, where many—perhaps, too many—states are merged, and the price one pays is that the merged states generate many new paths that are impossible in practice. Such impossible paths thwart validation of elementary safety properties.

State-space explosion arises in model construction for single sequential programs, as well. To limit state space, a compiler builds a naive program model—the program’s control-flow graph—which contains many more paths than what will actually be used during execution. The extra paths might well prevent validation of safety properties. Here is a contrived example: the control-flow graph for the program, `if even(x) then x:=succ x fi; if odd(x) then x:=0 else x:=1 fi`, contains two impossible paths, which prevent validation that the program must terminate with x equals 0.

Incremental Refinement

For these reasons, one wants a mechanism that incrementally refines a naive program model, eliminating impossible paths. We illustrate here a technique based on *filters* [20].

Say that we begin with a naive program model and try to validate that Φ holds true for all paths in the model. Perhaps the validation fails, because the naive model includes impossible paths that make Φ false. We want to refine the model—not abandon it—and repeat the attempt at validation. To do so, we define an additional abstract interpretation and apply it to the naive program model. We do so by *encoding the abstract interpretation as a proposition, Ψ , and we model check the formula, $\Psi \Rightarrow \Phi$, on every path of the naive model*. The abstract interpretation, coded as Ψ , filters out, on the fly, impossible paths.

This technique is reminiscent of a standard practice in model checking, that of attaching logical preconditions to strengthen the hypothesis of a formula to be proved (for example, limiting model search to just fair paths, e.g., $isFairPath \Rightarrow \Phi$ [7, 8]). But what is notable here is that an abstract interpretation, rather than a logical precondition, is attached. This technique has been used with success by Dwyer and Pasareanu [21] on a variety of problems in communicating systems.

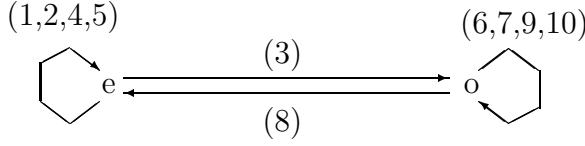
An abstract interpretation that uses a finite domain of abstract values can be encoded as a path proposition in this variant of LTL:

$$\Psi ::= \beta \mid X \mid \Psi \wedge \Psi \mid \neg \Psi \mid \langle A \rangle \Psi \mid \nu X. \Psi$$

Recall that LTL is logic of paths: for a path, p , say that $p \models \beta$ if state formula β holds true for p ’s start state; $p \models \langle A \rangle \Psi$ holds if $p = s_0 \xrightarrow{A} p_1$, and $p_1 \models \Psi$ holds (this is the “next state” modality); and $p \models \nu X. \Psi$ holds iff $p \models [\nu X. \Psi / X] \Psi$, that is, $\nu X. \Psi$ is a recursive proposition.

Characterizing Automata

To understand how to use LTL, we exploit the correspondence between finite-state automata and LTL, and present in Figure 8 an automaton that encodes even-odd abstract interpretation of a variable, x .



Automaton transitions:

- | | |
|---|---|
| (1) $\delta(e, \mathbf{E}) = e$, if $tt \in \llbracket \mathbf{E} \rrbracket e$ | (6) $\delta(o, \mathbf{E}) = o$, if $tt \in \llbracket \mathbf{E} \rrbracket o$ |
| (2) $\delta(e, \neg \mathbf{E}) = e$, if $ff \in \llbracket \mathbf{E} \rrbracket e$ | (7) $\delta(o, \neg \mathbf{E}) = o$, if $ff \in \llbracket \mathbf{E} \rrbracket o$ |
| (3) $\delta(e, \mathbf{x} := \mathbf{E}) = o$, if $o \in \llbracket \mathbf{E} \rrbracket e$ | (8) $\delta(o, \mathbf{x} := \mathbf{E}) = e$, if $e \in \llbracket \mathbf{E} \rrbracket o$ |
| (4) $\delta(e, \mathbf{x} := \mathbf{E}) = e$, if $e \in \llbracket \mathbf{E} \rrbracket e$ | (9) $\delta(o, \mathbf{x} := \mathbf{E}) = o$, if $o \in \llbracket \mathbf{E} \rrbracket o$ |
| (5) $\delta(e, \mathbf{y} := \mathbf{E}) = e$, if $\mathbf{y} \neq \mathbf{x}$ | (10) $\delta(o, \mathbf{y} := \mathbf{E}) = o$, if $\mathbf{y} \neq \mathbf{x}$ |

Both states are possible end states.

Fig. 8. Even-odd abstract interpretation encoded as a path formula

It is easy to imagine this automaton executed on the paths of a control-flow graph—as long as the automaton can continue to move while it traverses a path, the path is well-formed with respect to the even-odd-ness value of \mathbf{x} .¹⁷ The automaton “filters out” those paths in a naive model that are impossible with respect to \mathbf{x} ’s even-odd-ness, and in the process generates a more accurate, restricted, reduced model. (Consider the example program at the beginning of this section—the impossible paths are filtered out.)

Next, imagine the automaton executed on the naive model *in parallel* with a model check of a safety property, Φ . This would validate Φ just on those paths well formed with respect to \mathbf{x} ’s even-odd-ness. No intermediate, reduced model is built; the work is done on the naive model.

The automaton in the Figure is just a pictorial representation of this LTL formula: (For brevity, we limit the set of actions to just **even** \mathbf{x} , \neg **even** \mathbf{x} , $\mathbf{x} := \mathbf{x} + 1$, and $\mathbf{y} := 2$.)

$$\begin{array}{ll}
 isEven_x \text{ iff } \langle \mathbf{even } \mathbf{x} \rangle isEven_x & isOdd_x \text{ iff } \langle \neg \mathbf{even } \mathbf{x} \rangle isOdd_x \\
 \langle \mathbf{x} := \mathbf{x} + 1 \rangle isOdd_x & \langle \mathbf{x} := \mathbf{x} + 1 \rangle isEven_x \\
 \langle \mathbf{y} := 2 \rangle isEven_x & \langle \mathbf{y} := 2 \rangle isOdd_x \\
 isFinalState & isFinalState
 \end{array}$$

Thus, the parallel execution of even-odd analysis with the model checking of safety property Φ is just the model check of the formula

$$(isEven_x \Rightarrow \Phi) \wedge (isOdd_x \Rightarrow \Phi)$$

¹⁷ In fact, one simply traverses the synchronous product between the program model, which can be itself regarded as an automaton, and the characterizing automaton [15, 24].

Binding Time in Program Analysis

The method proposed above is based on the observation that program analysis can be refined by verifying implications on the logical level, which restricts the conditions under which a certain property must hold, or, equivalently, by verifying the original property for a product program model consisting of the original program model and a ‘filter’ automaton which restricts the behaviour of the program model to its desired fragment. This is possible, because the formulae considered as premises can be fully expressed in terms of automata. Based on this idea, it is now possible to choose

- where to put the complexity: in the formula or in the automaton,
- how to combine the product construction and the model checking.

Straightforward would be to first compute the product program model and subsequently apply the model checker, as suggested in the previous section. This would directly corresponds to the property-oriented expansion approach proposed in [49], which considers product construction with automata specified in different paradigms according to the unifying model idea presented in [50].

However, as also mentioned in the previous section, there are ‘on-the-fly’ model checkers allowing to construct the product program model on demand during the model checking procedure, which may help to avoid the state explosion problem. In fact, the same method can also be applied if one wants to verify a parallel program.

In this way, the line between abstract interpretation in model construction and property extraction via model checking has been blurred. The issue becomes one of “binding times”—when is the abstract interpretation “bound” into the program model? The answer often hinges on the desired complexity of the model versus the complexity of the formula to be model checked.

9 Conclusion

This paper has attempted to explain how abstract interpretation, flow analysis, and model checking can interact within a comprehensive static analysis methodology. In particular, we hoped to emphasize that the machinery and methods of flow analysis, abstract interpretation, and model checking have grown together, and that researchers can profitably use techniques from one area to improve results in the others.

In the future, it is planned to specifically support the study of method interaction, in order to improve the understanding of the interdependencies between and the ‘optimal’ problem-specific combinations of algorithms for e.g. model construction, abstraction and analysis as addressed here. A corresponding public platform is already available [46], and it is going to be used in program analysis [51].

Acknowledgements

We are grateful to Tiziana Margaria for proof reading, to Markus Müller-Olm for his constructive criticism, and to Andreas Holzmann for his support in the type setting.

References

1. P. Aczel. *Non-Well-Founded Sets*, Lecture Notes 14, Center for Study of Language and Information, Stanford, CA, 1988.
2. S. Bensalem and A. Bouajjani and C. Loiseaux and J. Sifakis. *Property preserving simulations*. Computer Aided Verification: CAV'92. Lecture Notes in Computer Science 663, Springer, 1992, 260–273.
3. D. Berry. *Generating Program Animators from Programming Language Semantics*, Ph.D. Thesis, LFCS Report ECS-LFCS-91-148, University of Edinburgh, 1991.
4. O. Burkart and B. Steffen. *Model Checking for Context-Free Processes*. Proceedings of the International Conference on Concurrency Theory, Concur95, LNCS 630, 1992
5. S. C. Cheung and J. Kramer. *An Integrated Method For Effective Behaviour Analysis of Distributed Systems*. Proceedings of the 16th International Conference on Software Engineering, Sorrento, CA, USA, 1994, pp. 309–320.
6. S. C. Cheung and J. Kramer. *Tractable Flow Analysis for Distributed Systems*. IEEE Transactions on Software Engineering 20-9 (1994).
7. E. Clarke and E. Emerson and A. Sistla. *Automatic verification of finite-state concurrent systems using temporal logic specifications*. ACM Transactions on Programming Languages and Systems 8 (1986) 244–263.
8. E.M. Clarke and O. Grumberg and D.E. Long. *Verification tools for finite-state concurrent systems*. In A Decade of Concurrency: Reflections and Perspectives, J.W. deBakker and W.-P. deRoever and G. Rozenberg", editors, Springer LNCS 803, 1993, pp. 124-175.
9. R. Cleaveland and P. Iyer and D. Yankelevich. *Optimality in abstractions of model checking*. Proc. SAS'95: Proc. 2d. Static Analysis Symposium, Lecture Notes in Computer Science 983, Springer, 1995, 1995.
10. R. Cleaveland, M. Klein and B. Steffen. *Faster Model Checking for the Modal μ -Calculus*. Proceedings of the International Workshop on Computer Aided Verification, CAV'92, LNCS 663, 1992
11. M. Codish and S. Debray and R. Giacobazzi. *Compositional analysis of modular logic programs*. Proc. 20th ACM Symp. on Principles of Programming Languages, 1993, pp. 451-464.
12. M. Codish and M. Falaschi and K. Marriott, *Suspension analysis for concurrent logic programs*. Proc. 8th Int'l. Conf. on Logic Programming, MIT Press, 1991, pp. 331-345.
13. G. Cousineau and M. Nivat. *On rational expressions representing infinite rational trees*. Proc. 8th Conf. Math. Foundations of Computer Science: MFCS'79, Lecture Notes in Computer Science 74, Springer, 1979, pp. 567–580.
14. P. Cousot, R. Cousot. *Abstract interpretation: A unified Lattice Model for static Analysis of Programs by Construction or Approximation of Fixpoints*. In Proceedings 4th ACM Symp. on Principles of Programming Languages, POPL'77, Los Angeles, California, January, 1977

15. P. Cousot and R. Cousot. *Systematic design of program analysis frameworks*. Proc. 6th ACM Symp. on Principles of Programming Languages, POPL'79, 1979, pages 269-282.
16. P. Cousot and R. Cousot. *Inductive Definitions, Semantics, and Abstract Interpretation*. Proc. 19th ACM Symp. on Principles of Programming Languages, POPL'92, 1992, pp. 83-94.
17. P. Cousot and R. Cousot. *Abstract interpretation frameworks*. Journal of Logic and Computation 2 (1992) 511-547.
18. D. Dams. *Abstract interpretation and partition refinement for model checking*. Ph.D. thesis, Technische Universiteit Eindhoven, The Netherlands, 1996.
19. M. Dwyer and L. Clark. *Data Flow Analysis for Verifying Properties of Concurrent Programs*. Proc. 2d ACM SIGSOFT Symposium on Foundations of Software Engineering, 1994, pp.62-75.
20. M. Dwyer and D. Schmidt, *Limiting State Explosion with Filter-Based Refinement*. Proc. International Workshop on Verification, Model Checking and Abstract Interpretation, Port Jefferson, Long Island, N.Y., <http://www.cis.ksu.edu/~schmidt/papers/filter.ps.Z>, 1997.
21. M. Dwyer and C. Pasareanu. *Filter-based Model Checking of Partial Systems*. Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Orlando, FL, USA, 1998.
22. E. Emerson, J. Lei, *Efficient model checking in fragments of the propositional mu-calculus*. In Proceedings LICS'86, 267-278, 1986
23. F. Giannotti and D. Latella, *Gate splitting in LOTOS specifications using abstract interpretation*. In Proc. TAPSOFT'93, M.-C. Gaudel and J.-P. Jouannaud, eds. LNCS 668, Springer, 1993, pp. 437-452.
24. Godefroid, P. and Wolper, P. *Using Partial orders for the efficient verification of deadlock freedom and safety properties*. Proc. of the Third Workshop on Computer Aided Verification, Springer-Verlag, LNCS 575, 1991, pp. 417-428.
25. M. Hecht, *Flow Analysis of Computer Programs*. Elsevier, 1977
26. N.D. Jones and C. Gomard and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
27. J. Kam and J. Ullman. *Global data flow analysis and iterative algorithms*. Journal of the ACM 23 (1976) 158-171.
28. G. A. Kildall. *A unified approach to global program optimization*. In Conf. Rec. 1st ACM Symposium on Principles of Programming Languages (POPL'73), pages 194 - 206. ACM, New York, 1973.
29. J. Knoop, B. Steffen and J. Vollmer *Parallelism for Free: Bitvector Analysis \Rightarrow No State explosion!* Proceedings of the International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'95, LNCS 1019, 1995
30. J. Knoop, O. Rüthing and B. Steffen. *Lazy Code Motion*. Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94), Orlando, Florida, SIPLAN Notices 30, 6 (1994), 233 - 245.
31. D. Kozen, *Results on the propositional mu-calculus*. Theoretical Computer Science, 27 (1983) 333-354.
32. Y.S. Kwong, *On reduction of asynchronous systems*. Theoretical Computer Science 5 (1977) 25-50.
33. S.P. Masticola and B.G. Ryder. *Static Infinite Wait Anomaly Detection in Polynomial Time*. Proceedings of ACM International Conference on Parallel Processing, 1990.

34. S.P. Masticola and B.G. Ryder. *A Model of Ada Programs for Static Deadlock Detection in Polynomial Time*. Proceedings ACM Workshop on Parallel and Distributed Debugging, 1991.
35. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
36. R. Milner and M. Tofte. *Co-induction in relational semantics*. Theoretical Computer Science, 17 (1992) 209-220.
37. A. Mycroft and N.D. Jones. *A relational framework for abstract interpretation*. In *Programs as Data Objects*, Lecture Notes in Computer Science 217, Springer, 1985, pp. 156-171.
38. F. Nielson, *A Denotational Framework for Data Flow Analysis*. Acta Informatica 18 (1982) 265-287.
39. K.M. Olender and L.J. Osterweil. *Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation*. IEEE Transactions on Software Engineering 16-3 (1990) 268-280.
40. K.M. Olender and L.J. Osterweil. *Interprocedural Static Analysis of Sequencing Constraints*. ACM Transactions on Software Engineering and Methodology 1-1 (1992) 21-52.
41. Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19, University of Aarhus, Denmark, 1981.
42. D.A. Schmidt, *Abstract interpretation of small-step semantics*. Proc. 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages, M. Dam and F. Orava, eds. Springer, 1996.
43. D.A. Schmidt, *Trace-based abstract interpretation of operational semantics*. *J. Lisp and Symbolic Computation*, 10 (1998) 237-271.
44. D.A. Schmidt, *Data-flow analysis is model checking of abstract interpretations*. Proc. 25th ACM Symp. on Principles of Prog. Languages, POPL98, 1998.
45. F. daSilva. *Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics*. Ph.D. thesis, LFCS report ECS-LFCS-92-241, Edinburgh University, Scotland, 1992.
46. B. Steffen, T. Margaria, V. Braun: *The Electronic Tool Integration platform: concepts and design*, [51] 1(1), pp. 9-30.
47. B. Steffen. *Data Flow Analysis as Model Checking*. Proceedings of the International Conference on Theoretical Aspects of Computer Software, TACS'91, LNCS 526, 1991
48. B. Steffen. *Generating Data Flow Analysis Algorithms from Modal Specifications*, International Journal on Science of Computer Programming, N. 21, 1993, pp. 115-139.
49. B. Steffen, *Property-oriented expansion*. Proc. Static Analysis Symposium: SAS'96, Lecture Notes in Computer Science 1145. Springer, 1996, pp. 22-41.
50. B. Steffen, *Unifying Models*. Proc. of the Annual Symposium on Theoretical Aspects of Computer Science, STACS'97, Lecture Notes in Computer Science 1200. Springer, 1997, pp. 1-20.
51. *Special Section on Programming Language Tools*, Int. Journal on Software Tools for Technology Transfer, Vol. 3, Springer Verlag, October 1998
52. A. Venet, *Abstract interpretation of the pi-calculus*. Proc. LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages, M. Dam and F. Orava, eds., LNCS 1192, Springer, 1996.
53. A. Venet, *Automatic Determination of Communication Topologies in Mobile Systems*. Proc. SAS'98, G. Levi, ed. Springer LNCS, 1998.

Certifying, Optimizing Compilation

Peter Lee

Carnegie Mellon University, Pittsburgh PA 15213, USA,

`petel@cs.cmu.edu`,

<http://www.cs.cmu.edu/~petel>

Abstract

In practice it is well-known that it is usually easier to check a proof than to generate one. Together with George Necula, I have made use of this idea to develop a mechanism that allows a host computer system to safely execute programs that are provided by an untrusted source. There are many examples of systems where a desire such a capability arises, including extensible operating systems, web servers, and run-time systems. In our approach, called *Proof-Carrying Code*, or simply PCC, the designer of the host system first makes a formal definition of a safety policy. Then, a programmer who wants to execute a program on the host system must provide the program in a special form that contains, addition to the native code, a formal proof of that the code adheres to the safety policy. The host can easily validate the proof using standard proof-checking techniques. If the validation succeeds, the code is guaranteed to respect the safety policy.

There are many benefits of PCC. Since the approach is based on a static verification of the program's safety, once validated, the program can be executed without relying on run-time checks. Hence it can be executed without expensive modification or interpretation. This guarantee holds even if the proof or native code are tampered with. Also, PCC can provide an important form of system security without using cryptography and without consulting any external trusted entities. Finally, safety is provided by a relatively small and simple proof-checking program, thereby enhancing the trustworthiness of the entire system.

While there is a significant engineering benefit to reducing the problem of code verification to a much simpler proof-checking process, this puts the application writer in the rather uncomfortable position of being obligated to generate formal proofs of programs. In order to automate this process for an important class of safety policies, we have developed and implemented the concept of a *Certifying Compiler*. A certifying compiler automatically generates the proof along with the target code from a given source program. The main complication in the construction of a certifying compiler is in the static program analyses; each analysis must provide enough information so that generating the proofs for the subsequently optimized code will always succeed.

This lecture describes the concepts of Proof-Carrying Code and Certifying Compilation, and gives a detailed overview of Touchstone, a certifying compiler for a safe subset of C. Particular attention will be paid to the reformulation of the program analyses, and how these affect the overall structure of the compiler.

Author Index

Albert, E., 262
Alpuente, Maria, 262

Bagnara, Roberto, 99
Bodei, Chiara, 168

Charatonik, Witold, 278

Degano, Pierpaolo, 168
Duggan, Dominic, 295

Eyssette, F., 311

Fages, François, 82
Falaschi, Moreno, 262
Faure, C., 311

Gallagher, John P., 246
Giacobazzi, Roberto, 215, 349
Gori, Roberta, 82
Gouranton, Valérie, 115
Goyal, Deepak, 327

Hagiya, Masami, 17
Handjieva, Maria, 200
Hill, Patricia M., 99
Hind, Michael, 57

Julián, P., 262

Knobe, Kathleen, 33

Lee, Peter, 381
Leuschel, Michael, 230
Levi, Francesca, 134

Paige, Robert, 327
Peralta, Julio C., 246
Pioli, Anthony, 57
Podelski, Andreas, 278
Priami, Corrado, 168

Ranzato, Francesco, 215
Rüthing, Oliver, 1

Sağlam, Hüseyin, 246
Sarkar, Vivek, 33
Schmidt, David, 351
Scozzari, Francesca, 215
Steffen, Bernhard, 351

Tadjouddine, M., 311
Tozawa, Akihiko, 17
Tzolovski, Stanislav, 200

Venet, Arnaud, 152
Vidal, German, 262
Volpe, Paolo, 184

Zaffanella, Enea, 99